Verifying cryptographic protocol implementations that use industrial cryptographic APIs

Gijs Vanspauwen
Bart Jacobs

Report CW 703, May 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Verifying cryptographic protocol implementations that use industrial cryptographic APIs

Gijs Vanspauwen
Bart Jacobs

Report CW 703, May 2017

Department of Computer Science, KU Leuven

Abstract

In this technical report we describe an approach for verifying cryptographic protocol implementations written in C. We statically prove the correctness of these implementations with the general purpose verifier VeriFast. More concretely we prove: memory safety, the absence of explicit and implicit information leaks, and functional correctness which includes protocol integrity. Our invariant-based approach requires an extension of the symbolic model of cryptography in order to work for protocol implementations in C written against an existing cryptographic API. Compared to the state of our work in March 2016, as described in TR CW694, we have significantly overhauled our approach, in order to remove a number of unsoundnesses as well as lift a number of limitations.

Verifying cryptographic protocol implementations
that use industrial cryptographic APIs

Gijs Vanspauwen and Bart Jacobs

imec-DistriNet, KU Leuven, 3001 Leuven, Belgium.

Abstract

In this technical report we describe an approach for verifying cryptographic protocol implementations written in C. We statically prove the correctness of these implementations with the general purpose verifier VeriFast. More concretely we prove: memory safety, the absence of explicit and implicit information leaks, and functional correctness which includes protocol integrity. Our invariant-based approach requires an extension of the symbolic model of cryptography in order to work for protocol implementations in C written against an existing cryptographic API. Compared to the state of our work in March 2016, as described in TR CW694, we have significantly overhauled our approach, in order to remove a number of unsoundnesses as well as lift a number of limitations.

1 Introduction

We almost cannot imagine our everyday lives anymore without a connection to the Internet. From managing your bank accounts to staying in touch with your friends: we heavily rely on this massive piece of technology. In order to provide a secure environment for these day-to-day activities, web browsers and the web servers they communicate with use cryptography.

In this technical report, we describe an approach to proving the correctness of such cryptographic software written in the C programming language. Our approach, depicted in Figure 1, allows to verify cryptographic protocol implementations. Such an implementation is a software realization of a particular communication pattern that establishes a certain security goal (e.g. the authentication of a request) using cryptographic primitives (e.g. key generation and encryption). Each protocol participant is assigned a specific role and different roles are implemented separately as, for instance, distinct C functions like the functions A and B from Figure 1. Together, these roles make up a protocol implementation. Assuming the cryptographic primitives are perfect, we prove that the implementation of such a protocol is memory safe, does not leak secrets and indeed achieves its security goals.

The approach we developed builds on top of the work from [9], and is similar to the approaches proposed in [2] and [6]: use a general purpose program verifier to verify protocol implementations that are written against a trusted API containing the cryptographic primitives. First, the semantics of the functions in this API are specified through contracts (i.e. the pre- and postconditions in Figure 1) which are assumed correct. Then, the corresponding verification methodology can use these contracts to reason about the protocol implementations (refinement type checking in [2] and symbolic execution in [6]).

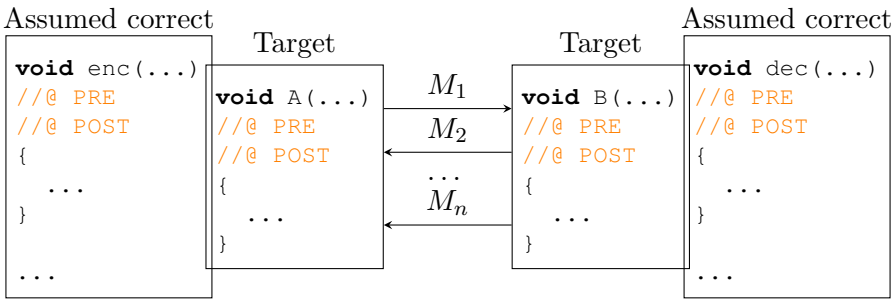


Figure 1: Overview of our approach for verifying cryptographic protocols

The cryptographic API in both [2] and [6] is designed in such a way that verification of the protocols within the symbolic model of cryptography is directly possible. In the symbolic model all messages are terms of a cryptographic algebra. The cryptographic primitives construct messages in this algebra and pairing/unpairing operators allow to compose/decompose messages. For this reason the cryptographic API in both [2] and [6] contains, besides the cryptographic primitives, functions to compose and decompose messages, and the network API accepts and returns these messages. The contracts of all the functions in the resulting API then allow to track the symbolic message content of memory regions during verification. More importantly however, the contracts of the network API enable the enforcement of a network invariant for messages allowed on the network. This network invariant is key to proving functional correctness and security of protocol implementations.

A crucial difference with our approach is that we target protocol implementations that employ preexisting cryptographic libraries. PolarSSL and the more widely known OpenSSL are two examples of such preexisting libraries that provide the required cryptographic functionality for writing protocol implementations and both contain a large protocol suite of their own. For now we focus on custom protocol implementations, but we see no reason that our approach would not work for preexisting protocol implementations as well. Since we target protocols that use preexisting libraries we do not have the liberty to design the trusted cryptographic API as we see fit. We are stuck with their accompanying APIs and most cryptographic libraries do not have a structured message concept such as is necessary for the approaches described in [2] and [6]. Instead, they leave it up to the protocol implementation to compose and parse messages using C buffers and C functions like `memcpy` and `memcmp`.

Our primary goal was to devise a technique that enabled us to verify protocol implementations that are written against a preexisting cryptographic API. Although in such a setting the symbolic model could not be directly applied as in [2] or [6], we aspired to use a similar symbolic reasoning. Therefore we introduced an extension of the symbolic model of cryptography. This extended model enables verification of protocol code that itself implements the composing and parsing of messages. It allows to associate symbolic cryptographic values with bit strings present in memory, and to keep track of these values while composing and parsing messages. In turn, the contracts of the functions in a cryptographic API can express associations between input and output buffers to functions on the one hand, and the symbolic result of cryptographic primitives on the other hand.

1.	$A \to B$:	$\textsc{AuthEnc}(k_x, m_x)$
2.	$B \to A$:	$\textsc{AuthEnc}(k_x, \{m_x, f(m_x)\})$

Figure 2: A confidential RPC protocol

The rest of this report starts with some background on invariant-based verification of cryptographic protocols within the symbolic model in Section 2. Next, we introduce our extended symbolic model in Section 3. Then, in Section 4, we show how to apply the extended model with a complete example. We give our results in Section 5 and finally we conclude this report in Section 6.

2 The Symbolic Model and Invariants

Before diving into the explanation of our extended symbolic model in Section 3, we first give some background on the verification of cryptographic protocols. In Section 2.1 we introduce a confidential Remote Procedure Call (RPC) protocol so we can immediately give concrete examples when we discuss different mechanism to state and show properties of protocols. Then, in Section 2.2, we discuss the symbolic model of cryptography. The symbolic model was first introduced in [5] as a tool to reason about protocols that used asymmetric encryption. It has since become a widespread instrument to reason about all kinds of cryptographic protocols. The simplicity of a term-based algebra together with some rules to define derivability of terms enables abstract reasoning about confidentiality properties without getting tangled up with implementation details. Another technique often used to state and show properties of cryptographic protocols are events or event predicates [2, 3, 4, 8]. We will call them events in Section 2.3 and correspondences between events allow to express integrity properties. A final mechanism we discuss in Section 2.4 to show protocol correctness are invariants [2, 3, 4, 6, 8]. More specifically we use network invariants to constrain what messages are allowed on the network and we call these public messages. We illustrate how to combine invariants with other introduced techniques in order to proof integrity and confidentiality properties.

2.1 A confidential RPC protocol

In Figure 2 the protocol transcript of a confidential Remote Procedure Call (RPC) protocol is shown that represents a single communication session. For a given confidential request message m_x, principal A wants to know the confidential response $f(m_x)$ for some function f that only principal B can compute.

As a first step in the protocol, principal A constructs the message $\textsc{AuthEnc}(k_x, m_x)$ by using the key k_x and the cryptographic primitive for authenticated symmetric encryption. Then principal A sends this message on the network for principal B to receive. After principal B has received this message, he uses the primitive for authenticated decryption to simultaneously check authenticity and retrieve m_x. Principal B then computes the result $f(m_x)$ and in turn constructs the message $\textsc{AuthEnc}(k_x, \{m_x, f(m_x)\})$. Next, principal A receives this message from B via the network. He also uses the authenticated decryption primitive to finally retrieve $f(m_x)$ and check the authenticity of this response. Principal A must also check that the response message $f(m_x)$ he got was indeed for the request message m_x and not actually $f(m)$ for some other request message m.

The intermediary steps taken by the protocol participants to correctly implement the confidential RPC protocol are left implicit in the protocol transcript, so protocol transcripts do not tell the full story. There are also some assumptions left implicit in the protocol transcript for this protocol to work such as the key k_x should be only known to principal A and B and both principals should keep it secret, and the freshness[1] of the request message m_x is not important to principal B.

2.2 The symbolic model

In the symbolic model of cryptography we identify any constructible message with an element t from some term algebra $\mathbb{T}$. If we denote with m an element from the set of raw messages $\mathbb{M}$ and with k a key from the set of keys $\mathbb{K}$, an example of such a term algebra is given by:

$$t \quad ::= \quad m \mid k \mid \{t, t\} \mid \textsc{AuthEnc}(k, t)$$

In the symbolic model one also introduces the notion of derivability of messages for the chosen term algebra[2]. A derivable term is a term known by the attacker and we will denote with $\lfloor t \rfloor$ that a term t is derivable by the attacker. Confidentiality of a message can then be expressed in terms of the corresponding term not being derivable. Assuming that the set $keys_{att}$ contains the keys initially known by the attacker and that the set $msgs_{att}$ contains the raw messages that are guessable by the attacker[3], the following rules partially define derivability of terms in our algebra:

$$\frac{k \in keys_{att}}{\lfloor k \rfloor} \qquad \frac{m \in msgs_{att}}{\lfloor m \rfloor} \qquad \frac{\lfloor \{t_1, t_2\} \rfloor}{\lfloor t_1 \rfloor} \qquad \frac{\lfloor \{t_1, t_2\} \rfloor}{\lfloor t_2 \rfloor} \qquad \frac{\lfloor t_1 \rfloor \qquad \lfloor t_2 \rfloor}{\lfloor \{t_1, t_2\} \rfloor}$$

$$\frac{\lfloor k \rfloor \qquad \lfloor t \rfloor}{\lfloor \textsc{AuthEnc}(k, t) \rfloor} \qquad \frac{\lfloor \textsc{AuthEnc}(k, t) \rfloor \qquad \lfloor k \rfloor}{\lfloor t \rfloor}$$

The attacker in the symbolic model has full control over the network. Full control means intercepting any message from the network and putting any message on the network that can be derived with terms known to the attacker. So the attacker can replace any message that is sent on the network during a protocol run with another. To take this into account the definition of attacker derivability is completed with a protocol specific rule for each message exchanged in a protocol run. This rule encodes the steps that a principal takes to construct the corresponding message and these steps may of course depend on earlier messages he found on the network which is under control of the attacker. To give a concrete example, here are the two rules that complete the definition of attacker derivability for the confidential RPC protocol from Section 2.1 given the key k_x, the request message m_x and the set $keys_B$ of keys known by principal B (so definitely $k_x \in keys_B$):

[1] An attacker can resend a request message he previously grabbed from the network and principal B has no means to detect this unless he records previous requests. If this situation is undesirable or can even cause a security breach in some particular application, the presented protocol is not suitable.

[2] This is often established with extra kinds of terms and equality axioms; for example introducing AuthDec terms requires an axiom saying decrypting an encrypted message with the same key, results in the original message: $\forall k, t.\ \textsc{AuthDec}(k, \textsc{AuthEnc}(k, t)) = t$. We chose not to have such terms and axioms, but to embed these properties directly into the definition of derivability.

[3] An implicit assumption here is that keys of honest principals and expected secrets of protocol runs are not members of the sets $keys_{att}$ or $msgs_{att}$ as the attacker can then trivially derive them.

$$\frac{}{\lfloor \textsc{AuthEnc}(k_x, m_x) \rfloor}\ [1.\ A \to B] \qquad \frac{\lfloor \textsc{AuthEnc}(k, m) \rfloor \qquad k \in keys_B}{\lfloor \textsc{AuthEnc}(k, \{m, f(m)\}) \rfloor}\ [2.\ B \to A]^4$$

Given this complete definition of attacker derivability for the confidential RPC protocol we can now already state two security goals concerning confidentiality given m_x:

- The request message m_x is confidential, i.e. $\lfloor m_x \rfloor$ is not derivable
- The response message $f(m_x)$ is confidential, i.e. $\lfloor f(m_x) \rfloor$ is not derivable

Both these properties are further on proven in Section 2.4.

2.3 Events and event correspondences

Another security goal of the confidential RPC protocol from Section 2.1, besides the confidentiality properties discussed in Section 2.2, is the integrity of both messages sent. With integrity of the first message we mean that principal B should only accept a request message from the network if principal A (or some other principal with whom B shares a key which we ignore here) indeed sent it. Integrity of the second message implies that A only accepts the response message if it is indeed a response from B to A's request.

Integrity properties can be expressed through correspondences between protocol events. Events are protocol-dependent and each event indicates that some principal has taken a specific step in the protocol run. A complete protocol run then generates a list or trace of events [8]. Event correspondences are then expressed through properties of all possible protocol traces. While other events could be identified, we define four custom events for the example protocol from Section 2.1:

$A_1^{\rightarrow}(m)$:	Principal A decided to send the request message m.
$B_1^{\leftarrow}(m)$:	Principal B accepted the request message m.
$B_2^{\rightarrow}(m)$:	Principal B decided to send the response message $f(m)$.
$A_2^{\leftarrow}(m)$:	Principal A accepted the response message $f(m)$.

The two event correspondences that denote the integrity of the example protocol are then:

$$\forall m.\, B_1^{\leftarrow}(m) \ \Rightarrow\ A_1^{\rightarrow}(m) \qquad\qquad \forall m.\, A_2^{\leftarrow}(m) \ \Rightarrow\ B_2^{\rightarrow}(m)$$

An intuitive account of the events for the confidential RPC protocol from Section 2.1 is shown in Figure 3. Consider an application running on host A that needs the computation f with m as input, but $f(m)$ can only be computed by an application running on host B. Both applications run within the corresponding application layers of the relevant host and the confidential RPC protocol itself is implemented in a lower layer of the (network) protocol stack on both hosts. The initial event $A_1^{\rightarrow}(m)$ can then be considered as an invocation of protocol role A for m by the application running on host A. The event $B_1^{\leftarrow}(m)$ in turn signals that the implementation of role B accepted the request message m as genuine and it unblocks the application on host B that was waiting for some m to compute the result $f(m)$. Once it has successfully calculated $f(m)$ the application continues the implementation of principal B by handing it the result $f(m)$ which corresponds to the event $B_2^{\rightarrow}(m)$. The last event

[4] We assume here that principal B is ready to respond to request messages from all the principals with whom he shares a key in the set $keys_B$. Moreover, we assume principal B knows which key to use on receiving a message from the network or that he tries all the keys from the set $keys_B$.

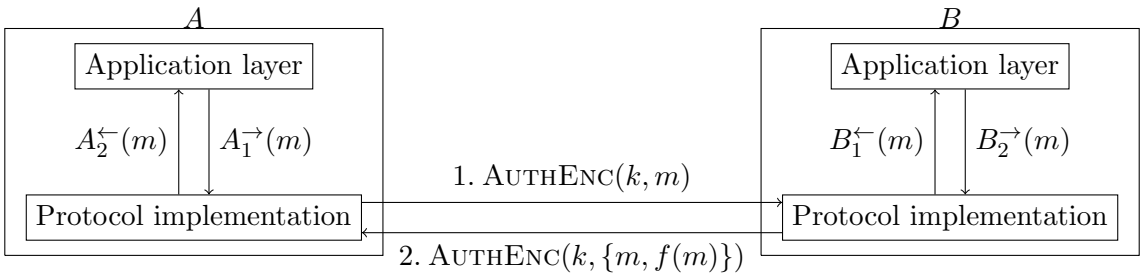


Figure 3: Overview of events and messages on the network for the protocol from Figure 2.

$A_2^{\leftarrow}(m)$ finally occurs when the implementation of role A accepts the response message $f(m)$ as genuine. The application on host A can now be unblocked, it receives the result $f(m)$ from the protocol implementation and continues its execution.

In order to define correct network invariants in Section 2.4, we restrict the set of possible events to contain only initial and final events. The occurrence of an initial event (e.g. $A_1^{\rightarrow}(m)$ or $B_2^{\rightarrow}(m)$) is determined before a protocol run starts and their occurrence is independent from the execution of the protocol. Initial events can be seen as preconditions for a protocol run to advance. Final events correspond to termination of a principal implementation (e.g. $B_1^{\leftarrow}(m)$ and $A_2^{\leftarrow}(m)$) so their occurrence depends on the execution steps taken by the protocol participants. Final events can be interpreted as postconditions that hold on successful termination of protocol participants.

Having defined protocol-specific events for the confidential RPC protocol, the protocol-specific part of the definition of attacker derivability has to be updated accordingly. Taking the initial events into account, the two rules below now complete the definition of attacker derivability given the key k_x:

$$\frac{A_1^{\rightarrow}(m)}{\lfloor \text{AuthEnc}(k_x, m) \rfloor}\ [1.\ A \to B] \qquad \frac{B_2^{\rightarrow}(m) \qquad \lfloor \text{AuthEnc}(k, m) \rfloor \qquad k \in keys_B}{\lfloor \text{AuthEnc}(k, \{m, f(m)\}) \rfloor}\ [2.\ B \to A]$$

The two confidentiality goals for the RPC protocol then become:

$$\forall m.\ A_1^{\rightarrow}(m)\ \wedge\ B_2^{\rightarrow}(m) \Rightarrow \neg \lfloor m \rfloor \qquad\qquad \forall m.\ A_1^{\rightarrow}(m)\ \wedge\ B_2^{\rightarrow}(m) \Rightarrow \neg \lfloor f(m) \rfloor$$

We have now dealt with initial events, but still need to express when the final events of the confidential RPC protocol occur. The following two statements encode when principal A, respectively principal B, is willing to accept a message from the network as genuine:

$$\forall k, m.\ \lfloor \text{AuthEnc}(k, m) \rfloor \wedge k \in keys_B \Rightarrow B_2^{\leftarrow}(m)$$
$$\forall m.\ \lfloor \text{AuthEnc}(k_x, \{m, f(m)\}) \rfloor \wedge A_1^{\rightarrow}(m) \Rightarrow A_2^{\leftarrow}(m)$$

2.4 Network invariants and public messages

In Section 2.2 we introduced attacker derivability as a way to state and reason about confidentiality properties. The definition of derivability as we presented it there is protocol-specific, implementation-dependent and can become quite complex for interesting protocols. Therefore, and to help us prove that some term is not derivable, we introduce an additional mechanism to help us reason about what the attacker knows: network invariants.

Within the symbolic model, a network invariant divides the terms of the chosen algebra into two disjoint sets: public terms and nonpublic terms. These invariants are protocol dependent and a good invariant ensures that all terms that must be published on the network in a protocol run are indeed public. This ensures that honest principals never violate the network invariant. Moreover the invariant must be closed under attacker actions; i.e. for any set of public terms, the attacker should only be able to derive new terms that are also public. This ensures that the attacker can never violate the network invariant. Once established that nobody can violate the chosen invariant, confidentiality of a term can then be proven by showing that the term is not public. This reasoning about derivability and public terms can be made more precise. For some chosen invariant I, first we must prove $\forall t.\, \lfloor t \rfloor \Rightarrow I(t)$ to show that I indeed allows protocol progress and is closed under attacker actions. Next, to prove the confidentiality of some term t_{sec}, it is sufficient to prove $\neg I(t_{sec})$.

Thus given the key k_x for a protocol run of the confidential RPC protocol from Section 2.1, a suitable invariant must ensure that all messages exchanged by principal A and B are public according to the invariant. For any proposed invariant I it must be so that $A_1^{\rightarrow}(m) \Rightarrow I(\textsc{AuthEnc}(k_x, m))$ and $B_2^{\rightarrow}(m) \Rightarrow I(\textsc{AuthEnc}(k_x, \{(m, f(m)\}))$. As mentioned before, a suitable invariant must also be closed under attacker actions. For the term algebra from Section 2.2 a proposed invariant I is closed under attacker actions if it satisfies:

$$\forall k \in keys_{att}.\, I(k) \qquad \forall m \in keys_{att}.\, I(m) \qquad \forall t_1, t_2.\, I(t_1) \wedge I(t_2) \Leftrightarrow I(\{t_1, t_2\})$$

$$\forall k, t.\, I(k) \wedge I(t) \Rightarrow I(\textsc{AuthEnc}(k, t))$$

Given the key k_x, we can now define a suitable invariant for some term t by recursion on t:

$$I_{RPC}(t) \equiv \begin{cases} m \in msgs_{att} & (\text{if } t = m) \\ k \in keys_{att} & (\text{if } t = k) \\ I_{RPC}(t_1) \wedge I_{RPC}(t_2) & (\text{if } t = \{t_1, t_2\}) \\ (I_{RPC}(k) \wedge I_{RPC}(t')) \vee & \\ \exists m.\, (A_1^{\rightarrow}(m) \wedge t' = m \wedge k = k_x) \vee & \\ \exists m, m_f.\, (B_2^{\rightarrow}(m) \wedge t' = \{m, m_f\} \wedge & \\ \qquad m_f = f(m) \wedge k \in keys_B)) & (\text{if } t = \textsc{AuthEnc}(k, t')) \end{cases}$$

Equipped with this invariant we can now finally prove the desired confidentiality and integrity properties of the confidential RPC protocol. We first show show that our chosen invariant is a useful one by proving Theorem 2.1. Theorem 2.2 then gives us the desired confidentiality properties and Theorem 2.3 the earlier stated integrity properties.

Theorem 2.1. *Derivable terms respect the chosen invariant:* $\forall t.\, \lfloor t \rfloor \Rightarrow I_{RPC}(t)$
Proof. For some t, induction on the derivation of $\lfloor t \rfloor$ allows to show $I_{RPC}(t)$ in all cases. □

Theorem 2.2. *Confidentiality property:* $\forall m.\, A_1^{\rightarrow}(m) \wedge B_2^{\rightarrow}(m) \Rightarrow \neg\lfloor m \rfloor \wedge \neg\lfloor f(m) \rfloor$
Proof. For any m assuming $A_1^{\rightarrow}(m) \wedge B_2^{\rightarrow}(m) \wedge (\lfloor m \rfloor \vee \lfloor f(m) \rfloor)$ and using Theorem 2.1 and the definition of $I_{RPC}(t)$ leads to a contradiction. □

Theorem 2.3. *Integrity property:* $\forall m.\, (B_1^{\leftarrow}(m) \Rightarrow A_1^{\rightarrow}(m)) \wedge (A_2^{\leftarrow}(m) \Rightarrow B_2^{\rightarrow}(m))$
Proof. For any m assuming $B_1^{\leftarrow}(m) \wedge \neg A_1^{\rightarrow}(m)$ and using Theorem 2.1 and the definition for $B_1^{\leftarrow}(m)$ and $I_{RPC}(t)$ leads to a contradiction. Similarly, assuming $A_2^{\leftarrow}(m) \wedge \neg B_2^{\rightarrow}(m)$ leads to a contradiction. □

3 The Extended Symbolic Model in VeriFast

Here, we discuss our extended symbolic model of cryptography which is based on the symbolic model and its associated techniques discussed in Section 2. Since we encoded this model directly in VeriFast[5] (a formal definition with a soundness proof in the random oracle model [1] is currently being developed), it is instructive to familiarize yourself with general verification in VeriFast before you continue reading [7, 10]. The encoding of our extended symbolic model depends on various concepts from this general purpose verifier for C programs. So in the rest of this text, basic knowledge about verification with VeriFast is assumed.

In the outline of our approach here we focus on the generation of random values (i.e. symmetric keys and random nonces), hashing and symmetric authenticated encryption. Since the semantics of regular symmetric encryption are inherently more complex, we postpone its discussion to Appendix A. Our approach deals with asymmetric encryption and signatures in a very similar way as regular symmetric encryption and they are not discussed in this text. The full encoding of our extended symbolic model in VeriFast is available in the `examples/crypto_ccs` directory of the latest VeriFast release. This encoding also deals with keyed hashes, asymmetric encryption and signatures.

Now, we will introduce our extended symbolic model of cryptography step-by-step. We start by showing a template for a verified protocol in Section 3.1. While this template does not yet introduce any concepts or definitions from our extended symbolic model, it will be useful while explaining later definitions. Then, in Section 3.2, we give the definitions to track principal identities during symbolic execution. Each function implementing (part of) a protocol role will need such an identity as a permission to invoke specific other functions from the cryptographic API (e.g. for generating a random value). Next we show how verified protocol code can actually send bytes on the network in Section 3.3. For this we chose the fairly standard (i.e. comparable to the `sys/socket.h` header from POSIX) network API of PolarSSL[6] and augmented it with VeriFast contracts to specify its semantics. Note that any other network API for C could be used as well provided it is augmented with analogous contracts. Subsequently in Section 3.4 we discuss how we represent cryptographic information; i.e. memory contents produced by the cryptographic primitive implementations. This is a fundamental aspect of our extended symbolic model. The representation of cryptographic information ensures that the caller of a cryptographic primitive does not automatically have the permission to read from the corresponding memory region. This prevents secret information from leaking into regular program variables or into the program's control flow. Section 3.4 also discusses that to get read permissions to such a memory region, one has to prove that it does not contain any secrets. In Section 3.5 then, we describe how we symbolically represent the result of the cryptographic primitives. We call these symbolic results cryptograms and they are essential to give meaningful contracts to the relevant PolarSSL cryptographic primitive implementations in Section 3.6. The rationale for choosing PolarSSL here is that it is a lightweight, and thus relatively simple cryptographic library as it targets the embedded market. Again a different cryptographic API for C could be used instead, as long as the contracts are analogous to the ones shown in Section 3.6. How to prove that a memory region directly produced by some cryptographic primitive does not contain any secrets in order to get read

[5] VeriFast: `https://github.com/verifast/verifast`

[6] PolarSSL (recently rebranded to mbed TLS): `https://tls.mbed.org/`

permissions to it, is explained in Section 3.7. We use an invariant-based approach for these proofs as introduced in Section 2, very similar to [2] and [6]. The main difference is that we enforce this invariant on cryptographic information in readable memory regions, instead of on messages on the network. These invariants as in Section 2 are also the key mechanism to prove security properties of protocol implementations. Our attacker model then, is defined in Section 3.8 and finally, we conclude in Section 3.9 with a discussion of an induction principle for cryptograms which allows the verification of recursive cryptographic protocols.

3.1 Verified protocol template

A template for a two-party protocol is shown in Listing 1. This template does not yet introduce any concepts or definitions from our extended symbolic model itself and it is completely understandable with the explanations from the VeriFast Tutorial [7]. It will however, be very helpful in explaining all the definitions that follow. We will gradually explain how to fill in this template in order to end up with a verified protocol.

The two roles from the protocol template from Listing 1 are implemented as indefinitely repeated distinct threads of the same application on the same host. However, after having verified a filled in template within our extended symbolic model, the two roles can be distributed over different hosts without violating the properties proven during verification and they will be secure under our attacker model and multiple concurrent runs. To fill in this template, the code that implements the different protocol roles should be placed in the two functions `role1` and `role2`. If this code requires access to some specific resources, the argument lists of `role1` and `role2` and also their contracts have to be updated accordingly. The updated contracts of both these functions should not only give the implementing code the necessary permissions, it should also encode the security properties of the protocol at hand. How to encode these properties is gradually explained throughout this text.

Before we continue the discussion of the template from Listing 1, it is instructive to look at a small example of a partially filled in template. Listing 2 shows such an example where `role1` requires an argument `arg` and the permission `perm`. The example illustrates that for each role the bodies of the predicate instances `pthread_run_pre` and `pthread_run_post` should be updated to reflect the chosen argument list and contract for the corresponding C function. Also the call to function `role1` in `role1_t` has to be adjusted accordingly. Both the predicate instances `pthread_run_pre` and `pthread_run_post` and the function `role1_t` are defined here because of the way that VeriFast deals with threading and are not further discussed (see [7] for more details).

After launching the attacker thread, the `main` function from the template in Listing 1 invokes both protocol roles as different threads inside an infinite loop. The loop does not wait for these threads to join in order to allow multiple concurrent executions of the protocol and it uses two **`leak`** statements to cleanup the symbolic heap. Finally, the last step in filling out this template is to set up the right context (which will be protocol-dependent, e.g. generating a key shared among protocol roles) in the loop of the `main` function before launching the different threads. After all these steps (not necessarily in discussed order), and after having verified the filled in template, the result is a verified cryptographic protocol implementation.

```
void role1()
  //@ requires true;
  //@ ensures true;
{ /* ... */ }

/*@
  predicate_family_instance pthread_run_pre(role1_t)
                            (void *data, any info) = info == none;
  predicate_family_instance pthread_run_post(role1_t)
                            (void *data, any info) = info == none;
@*/

void *role1_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(role1_t)(data, ?x);
  //@ ensures pthread_run_post(role1_t)(data, x) &*& result == 0;
{
  //@ open pthread_run_pre(role1_t)(data, _);
  role1();
  //@ close pthread_run_post(role1_t)(data, _);
  return 0;
}

void role2()
  //@ requires true;
  //@ ensures true;
{ /* ... */ }

/*@
  predicate_family_instance pthread_run_pre(role2_t)
                            (void *data, any info) = info == none;
  predicate_family_instance pthread_run_post(role2_t)
                            (void *data, any info) = info == none;
@*/

void *role2_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(role2_t)(data, ?x);
  //@ ensures pthread_run_post(role2_t)(data, x) &*& result == 0;
{
  //@ open pthread_run_pre(role2_t)(data, _);
  role2();
  //@ close pthread_run_post(role2_t)(data, _);
  return 0;
}

int main(void)
  //@ requires true;
  //@ ensures  true;
{
  // ... create a thread that executes the attacker implementation

  while (true)
    //@ invariant true;
  {
    pthread_t t1, t2;
    //@ close pthread_run_pre(role1_t)(NULL, none);
    //@ close pthread_run_pre(role2_t)(NULL, none);
    pthread_create(&t1, NULL, &role1_t, NULL);
    pthread_create(&t2, NULL, &role2_t, NULL);
    //@ leak pthread_thread(_, role1_t, NULL, none);
    //@ leak pthread_thread(_, role2_t, NULL, none);
  }
}
```

Listing 1: A template for a verified two-party protocol

```
//@ predicate perm() = true /* &*& ... */;

void role1(int arg)
  //@ requires perm();
  //@ ensures perm();
{ /* ... */ }

struct args{ int arg; };

/*@
  predicate_family_instance pthread_run_pre(role1_t)(void *args, any info) =
    args_arg(args, ?val) &*& info == some(val) &*& perm();
  predicate_family_instance pthread_run_post(role1_t)(void *args, any info) =
    args_arg(args, ?val) &*& info == some(val) &*& perm();
@*/
void *role1_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(role1_t)(data, ?x);
  //@ ensures pthread_run_post(role1_t)(data, x) &*& result == 0;
{
  //@ open pthread_run_pre(role1_t)(data, _);
  role1(((struct args*) data)->arg);
  //@ close pthread_run_post(role1_t)(data, _);
  return 0;
}
```

Listing 2: Example of a filled in template for one protocol role

3.2 Tracking identities of principals

As part of our model, each function that implements a protocol role will need an identity to invoke specific functions from the trusted cryptographic API. This identity is not only a permission to generate random values (see Section 3.6) or to perform unauthenticated decryption (see Appendix A), it also allows us to link generated random values to the identity of the creator. During the verification of, according to our model, misbehaving protocol code, a permission can be revoked in order to prevent the code from unnoticeably undermining its own security goals. This mechanism is explained in Appendix A.

The definitions for tracking principal identities during verification are shown in Listing 3. For clarity, the different permissions that an identity comprises are defined separately as the predicates `random_permission` and `decryption_permission`. A principal identity then is defined in terms of these permissions, as a chunk of the predicate `principal`. Its first argument is the sequence number in the line of generated identities to ensure uniqueness. The second argument keeps track of how many random values are generated for the principal. The predicate `principals` is used to keep track of how many identities are generated thus far as indicated by its only argument. To retrieve the permission for generating identities (i.e. a chunk of the predicate `principals`), the lemma `principals_init` should be invoked. The precondition of this lemma uses the module system of VeriFast (notice the **`require_module`** declaration) to ensure that the lemma can only be invoked once in order to prevent the recycling of identities. How exactly this module system works is not explained here, but illustrative examples can be found in the latest VeriFast release. With the permission for generating identities, one can start creating new identities by invoking the lemma `principal_create`.

```
/*@
predicate decryption_permission(int principal;);
predicate random_permission(int principal, int generated_values;);

predicate principal(int principal, int generated_values;) =
  decryption_permission(principal) &*&
  random_permission(principal, generated_values)
;

predicate principals(int count);

require_module principals_mod;
lemma void principals_init();
  requires module(principals_mod, true);
  ensures  principals(0);

lemma int principal_create();
  requires principals(?count);
  ensures  principals(count + 1) &*& result == count + 1 &*&
           principal(count + 1, 0);
@*/
```

Listing 3: Tracking principal identities and corresponding permissions

Listing 4 illustrates how to use these definitions by showing a fragment of the template from Section 3.1 that has been partially filled in. The contract of `role1` reflects that this implementation of a protocol role needs access to a principal identity. Important to note here is that each piece of protocol code should only have access to one principal identity. Otherwise a misbehaving piece of protocol code which gets some of its identity permissions revoked (e.g. to prevent further usage of unauthenticated decryption, see Appendix A), can still fall back to another identity. This would make our encoding of the extended symbolic model in VeriFast unsound and is therefore not allowed in our approach. The `main` function in Listing 4 gets the permission to generate identities by invoking the lemma `principals_init` and generates two principal identities before launching the protocol. After the protocol has finished the identity permissions are cleaned up through **`leak`** statements.

3.3 The network API

An extract from the annotated network API is shown in Listing 5. The C functions shown are obtained from the network API of PolarSSL and they form a classical socket API. Specifications added to these C functions ensure that verified code uses them in a correct fashion. For brevity, only the C functions to establish a connection at client-side are shown. The functions and contracts for server-side connections are analogous.

To establish a connection, a client first has to call the `net_connect` function with the correct arguments. If successful (i.e. the result of the call is equal to zero), the client receives a chunk of the predicate `net_status`. As the postcondition of `net_connect` indicates the third argument of this chunk will be `connection_init`. This reflects the fact that the initialization of the socket is not complete after only calling `net_connect`, since the communication type still has to be set to blocking or non-blocking. So the final step in initializing a client socket is calling `net_set_block`. For simplicity we only support blocking communication, but non-

```
//@ import_module principals_mod;
void role1()
  //@ requires principal(?p1, ?random_values);
  //@ ensures  principal(p1, ?new_random_values);
{
  //@ open principal(p1, random_values);
  /* ... */
  //@ close principal(p1, _);
}

int main(void)
  //@ requires module(template, true);
  //@ ensures  true;
{
  //@ open_module();
  //@ principals_init();

  // ... create a thread that executes the attacker implementation

  while (true)
    //@ invariant principals(_);
  {
    //@ int p1 = principal_create();
    //@ int p2 = principal_create();

    // ... launch protocol role threads
    //@ leak principal(p1, _);
    //@ leak principal(p2, _);
  }
}
```

Listing 4: Illustration of using the definitions for principal identities

```
/*@
inductive socket_status =
 | bound_to_port | connection_init | connected;

predicate net_status(int socket, list<char> address,
                     int socket_port, socket_status status);
@*/
int net_connect(int *socket, const char *host, int port);
  //@ requires integer(socket, _) &*& [?f]option_string(host, ?h);
  /*@ ensures  integer(socket, ?socket_v) &*&
               [f]option_string(host, h) &*&
               result != 0 ? true :
               net_status(socket_v, h, port, connection_init); @*/

int net_set_block(int socket);
  //@ requires net_status(socket, ?h, ?port, connection_init);
  /*@ ensures  result != 0 ? true :
               net_status(socket, h, port, connected) ; @*/

int net_send(void *socket, const char *buf, size_t len);
  /*@ requires integer(socket, ?socket_v) &*&
               net_status(socket_v, ?ip, ?port, connected) &*&
               len <= MAX_MESSAGE_SIZE &*&
               [?f1]chars(buf, len, ?cs); @*/
  /*@ ensures  integer(socket, socket_v)  &*&
               net_status(socket_v, ip, port, connected) &*&
               [f1]chars(buf, len, cs); @*/

int net_recv(void *socket, char *buf, size_t len);
  /*@ requires integer(socket, ?socket_v) &*&
               net_status(socket_v, ?ip, ?port, connected) &*&
               chars(buf, len, _) &*& len <= MAX_MESSAGE_SIZE; @*/
  /*@ ensures  integer(socket, socket_v)  &*&
               net_status(socket_v, ip, port, connected) &*&
               chars(buf, len, _) &*& result <= len; @*/

void net_close(int socket);
  //@ requires net_status(socket, _, _, _);
  //@ ensures  true;
```

Listing 5: Extract from the annotated network API

```
void role1()
  //@ requires principal(?p1, ?random_values);
  //@ ensures principal(p1, ?new_random_values);
{
  int socket;
  char buffer[16];
  if(net_connect(&socket, NULL, 1234) != 0) abort();
  if(net_set_block(socket) != 0) abort();
  /* ... */
  net_send(&socket, buffer, 16);
  net_recv(&socket, buffer, 16);
  net_close(socket);
}
```

Listing 6: Illustration of how to use the network API

blocking communication could be easily added. After a successful call to `net_set_block` the client receives a chunk of the form `net_status(_, _, _, connected)` and he can start sending and receiving messages via the corresponding socket using the functions `net_send` and `net_receive`. The contracts of these functions require the caller to have a correctly initialized `socket`, i.e. a chunk of the form `net_status(socket, _, _, connected)`. The actual messages sent and received are simply character buffers. Finally, the function `net_close` allows one to close a socket.

An example of how to use these functions is shown in Listing 6. A client sets up a connection with a server at localhost (the default address if the NULL address is given) on port 1234. Then, he sends and receives a 16 byte message before closing the connection.

3.4 Representation of cryptographic information

Here we will discuss a fundamental aspect of our extended model of cryptography: the representation of cryptographic information generated by the cryptographic primitives and what is considered to be public and secret in our model. We verify protocols within the random oracle model (ROM) [1] where the cryptographic primitives are considered to be random oracles and in Section 3.4.1 we define the type `crypto_char` of cryptographic bytes to represent their results. Of course a concrete C implementation of a primitive must write its results somewhere in memory. How to specify the possibly secret cryptographic information in memory produced by a primitive is the subject of Section 3.4.2. There, we define the predicate `crypto_chars` as a cryptographic analogue of the predicate `chars` discussed in the VeriFast tutorial [7]. However, a memory region described by a `crypto_chars` chunk cannot be read until it is converted to a regular `chars` chunk. When such a conversion is allowed is explained in Section 3.4.3. In Section 3.4.4 we present a way to compose possibly secret memory regions. This is necessary when a conversion is not allowed for some `crypto_chars` chunk, but it needs to be merged with another such chunk. Finally, Section 3.4.5 discusses how to explicitly clear secrets described by a `crypto_chars` chunk from memory.

3.4.1 A type for cryptographic bytes

We verify protocols within the random oracle model (ROM) [1] and so contracts for the cryptographic primitives should take this into account. In the ROM cryptographic primitives like hashing and encryption are considered to be deterministic random oracles. So not only the generation of random keys, but all cryptographic primitives are considered to produce random results. As in the ROM we can assume that all randomness stems from a single source and we model this source with a finite[7] list of coin tosses. Some results of the cryptographic primitives will be public and others secret, so we can conceptually partition the finite list of coin tosses into a public part and a secret part. One of the main objectives of our approach is to prevent the leakage of bytes that depend on the secret coin tosses into regular program variables or into the program's control flow[8]. Therefore we will define a VeriFast type for the cryptographic bytes produced by the primitives that may depend on the secret coin tosses and treat values of this type in a special way.

First, we define the abstract ghost type `secret_coin_tosses` from Listing 7 as the type of finite lists of secret coin tosses. Important to note is that for a specific symbolic execution the actual list of secret coin tosses itself is not explicitly determined. Its main purpose is to differentiate between public and possibly secret bytes and an instance is never needed explicitly. Then, we can introduce the new type `crypto_char` from Listing 7 for cryptographic bytes. In VeriFast a specific byte in memory is normally represented in annotations with the type **`char`**. Within the setting of verifying cryptographic protocols values of the type **`char`** do not depend on the secret coin tosses, but values of the type `crypto_char` thus can. The definition of this type makes the dependency on the implicit secret coin tosses explicit. Indeed, a value of the type `crypto_char` corresponds directly to a function that takes a list of secret coin tosses and returns some value of the type **`char`**.

The next two definitions from Listing 7 allow to lift a value of the type **`char`** to a value of the type `crypto_char`. The auxiliary function `crypto_char_const` ignores the list of secret coin tosses given as its second argument and immediate returns the value of type **`char`** given as its first. So for a given value `c`, the expression `(crypto_char_const)(c)` is of the type **`fixpoint`**`(secret_coin_tosses, char)` which is exploited by the function `c_to_cc` to construct a value of the type `crypto_char`. The function `cs_to_ccs` finally, lifts a value of the type `list<`**`char`**`>` to a value of the type `list<crypto_char>`.

3.4.2 Memory regions containing cryptographic bytes

As already mentioned, the result of a cryptographic primitive could be a secret and we want to prevent leaking secrets to the attacker. For this reason we are going to treat C buffers where cryptographic primitives store their results in a special way. Permissions of C buffers are normally tracked by chunks of the predicate `chars` as declared in Listing 8. A chunk `chars(buffer, n, cs)` for example, indicates that there is a valid allocated memory region starting at the address `buffer` with a size of `n` bytes and with content `cs` of type `list<`**`char`**`>`. The owner of such a chunk has the permission to read and write the indicated memory region.

[7] We only consider finite prefixes of possibly non-terminating runs, so a finite source of randomness suffices.
[8] This ensures that protocol executions that agree on the public coin tosses follow the same execution path.

```
/*@
abstract_type secret_coin_tosses;

inductive crypto_char =
  cc_constructor(fixpoint(secret_coin_tosses, char) cc_function)
;

fixpoint char crypto_char_const(char c, secret_coin_tosses oracle)
{
  return c;
}

fixpoint crypto_char c_to_cc(char c)
{
  return cc_constructor((crypto_char_const)(c));
}

fixpoint list<crypto_char> cs_to_ccs(list<char> cs)
{
  switch(cs)
  {
    case cons(c, cs0):
      return cons(c_to_cc(c), cs_to_ccs(cs0));
    case nil:
      return nil;
  }
}
@*/
```

Listing 7: A type for cryptographic bytes: crypto_char

```
/*@
predicate chars(char *buffer, int n; list<char> cs);

inductive crypto_chars_kind = normal | secret;

predicate crypto_chars(crypto_chars_kind kind, char *buffer,
                       int n; list<crypto_char> ccs);
@*/
```

Listing 8: The predicates `chars` and `crypto_chars`

We now introduce the very similar predicate `crypto_chars` also from Listing 8. The difference with the `chars` predicate is that memory regions expressed by such a `crypto_chars` chunk contain possibly secret cryptographic information and such a memory region is neither readable nor writable directly. Moreover, as a memory region described by a `crypto_chars` chunk could be generated by a cryptographic primitive, it can depend on the implicit secret coin tosses and so its content is described by a value of the type `crypto_char`.

Chunks of the predicate `crypto_chars` come in two flavors and what flavor a chunk belongs to depends on its first argument. A `crypto_chars` chunk where the first argument is `normal` is equivalent to a regular `chars` chunk. This means it is not dependent on the secret coin tosses, not confidential and thus can be converted to a `chars` chunk and back with the lemmas `crypto_chars_to_chars` and `chars_to_crypto_chars` from Listing 9 respectively. Only when it is converted to a `chars` chunk does one obtain read and write permissions to the corresponding memory region. Note the usage of the pure function `cs_to_ccs` from Listing 7 in the contracts of both functions.

In normal circumstances (i.e. if no cryptographic collision occurs), a `crypto_chars` chunk where the first argument is `secret` can only be converted to a `chars` chunk after one has proven that its content is not secret. This important restriction prevents actual secret information from leaking into regular program variables or into the program's control flow. Such a `crypto_chars(secret, _, _, _)` chunk can also be converted to a `chars` chunk with the lemma `crypto_chars_to_chars` if a cryptographic collision occurs (i.e. if `col` equals **`true`**). This poses no problem as we prove all our protocol properties "up to a collision", meaning that the proven properties only hold for a specific protocol run if no cryptographic collision occurred (see further). The final lemma from Listing 9, `chars_to_secret_crypto_chars`, allows to convert any `chars` chunk to a `crypto_chars(secret, _, _, _)` chunk.

3.4.3 Proving a possible secret is not secret

A value `cs` of the type `list<char>` definitely does not depend on the secret coin tosses, but a value `ccs` of type `list<crypto_char>` can. So what if we can show in some symbolic execution branch that the value `ccs` is equal to `cs_to_ccs(cs)`? Then this is of course true for all possible choices of the secret coin tosses and thus `ccs` effectively does not depend on them. This reasoning is captured by the definitions from Listing 10.

The definition of predicate `public_ccs` from Listing 10 expresses that for the given value `ccs` there exists some value `cs` of type `list<char>` such that `ccs == cs_to_ccs(cs)`. Then lemma `public_crypto_chars` allows to convert any `crypto_chars` chunk (thus also `secret` ones) to a `chars` chunk if it is given a proof that the corresponding value of type `list<crypto_char>` does not depend on the secret coin tosses.

```
/*@
fixpoint bool col();

lemma_auto void crypto_chars_to_chars(char *array, int n);
  requires [?f]crypto_chars(?kind, array, n, ?ccs) &*&
           col || kind == normal;
  ensures  [f]chars(array, n, ?cs) &*& ccs == cs_to_ccs(cs);

lemma_auto void chars_to_crypto_chars(char *array, int n);
  requires [?f]chars(array, n, ?cs);
  ensures  [f]crypto_chars(normal, array, n, cs_to_ccs(cs));

lemma_auto void chars_to_secret_crypto_chars(char *array, int n);
  requires [?f]chars(array, n, ?cs);
  ensures  [f]crypto_chars(secret, array, n, cs_to_ccs(cs));
@*/
```

Listing 9: Some conversions between chars and crypto_chars

```
/*@
predicate public_ccs(list<crypto_char> ccs) =
  [_]exists(?cs) &*& ccs == cs_to_ccs(cs);

lemma void public_crypto_chars(char *array, int n);
  requires [?f]crypto_chars(_, array, n, ?ccs) &*& [_]public_ccs(ccs);
  ensures  [f]chars(array, n, ?cs) &*& ccs == cs_to_ccs(cs);
@*/
```

Listing 10: Converting a `crypto_chars` chunk to a `chars` chunk

```
/*@
lemma void crypto_chars_split(char *array, int i);
  requires [?f]crypto_chars(?kind, array, ?n, ?ccs) &*&
           0 <= i &*& i <= n;
  ensures  [f]crypto_chars(kind, array, i, ?ccs1) &*&
           [f]crypto_chars(kind, array + i, n - i, ?ccs2) &*&
           ccs1 == take(i, ccs) &*& ccs2 == drop(i, ccs) &*&
           ccs == append(ccs1, ccs2);

lemma_auto void crypto_chars_join(char *array);
  requires [?f]crypto_chars(?kind, array, ?n1, ?ccs1) &*&
           [f]crypto_chars(kind, array + n1, ?n2, ?ccs2);
  ensures  [f]crypto_chars(kind, array, n1 + n2, append(ccs1, ccs2));
@*/
```

Listing 11: Splitting and joining possibly confidential memory regions

```
void memcpy(void *dst, void *src, size_t count);
  /*@ requires crypto_chars(_, dst, count, _) &*&
               [?f]crypto_chars(?kind, src, count, ?ccs); @*/
  /*@ ensures  crypto_chars(kind, dst, count, ccs) &*&
               [f]crypto_chars(kind, src, count, ccs); @*/
```

Listing 12: A contract for `memcpy`

3.4.4 Composing possibly secret memory regions

In the VeriFast Tutorial [7] it is explained that two separately tracked adjacent character buffers can be merged together with a lemma called `chars_join` and any tracked character buffer can be split into two with the lemma `chars_split`. Listing 11 shows the `crypto_chars` counterparts of these lemmas. A first step in creating a single memory region from different arbitrary parts then, is to create `crypto_chars` chunks for the desired memory regions with the lemmas `crypto_chars_split` and `crypto_chars_join`. The next step is to copy these parts to adjacent regions in memory before they can be merged with the lemma `crypto_chars_join` as a final step. In many C programs, and also in our approach, this copying is done with the function `memcpy` from the C standard library. Listing 12 shows a contract for `memcpy` that can handle `crypto_chars` chunks. This contract simply states that after a call to `memcpy`, the content of the input buffer `src` is copied to the output buffer `dst` and the output buffer is described by the same kind of chunk as the input buffer was described with before the call.

Listing 13 shows a small verified example that uses all these definitions. In this example, the content of a non-confidential buffer of `SIZE` bytes at location `fst` is concatenated with the first `SIZE` bytes of a possibly confidential buffer at location `snd`. The result of this concatenation is written in the buffer `out`. While most of this example speaks for itself, the first statement invokes the lemma `chars_limits` and is required to assure VeriFast that no overflow occurs in the expression `out + SIZE`. More details on this can be found in [7].

```
void compose(char* fst, char* snd, char* out)
  /*@ requires chars(fst, SIZE, ?cs1) &*&
               crypto_chars(?kind, snd, 2 * SIZE, ?ccs2) &*&
               chars(out, 2 * SIZE, _); @*/
  /*@ ensures  chars(fst, SIZE, _) &*&
               crypto_chars(kind, snd, 2 * SIZE, ccs2) &*&
               crypto_chars(kind, out, 2 * SIZE,
                            append(cs_to_ccs(cs1), take(SIZE, ccs2))); @*/
{
  //@ chars_limits(out);
  //@ chars_to_crypto_chars(out, SIZE);
  //@ chars_to_crypto_chars(out + SIZE, SIZE);

  //@ chars_to_crypto_chars(fst, SIZE);
  memcpy(out, fst, SIZE);
  //@ crypto_chars_to_chars(fst, SIZE);
  /*@ switch(kind)
      {
        case secret:
          crypto_chars_to_chars(out, SIZE);
          chars_to_secret_crypto_chars(out, SIZE);
        case normal:
      }
  @*/
  //@ crypto_chars_split(snd, SIZE);
  memcpy(out + SIZE, snd, SIZE);
  //@ crypto_chars_join(snd);
  //@ crypto_chars_join(out);
}
```

Listing 13: Example using definitions from Listing 11 and Listing 12

```
void zeroize(char *buffer, int size);
  //@ requires crypto_chars(_, buffer, size, _);
  //@ ensures  chars(buffer, size, _);
```

Listing 14: A function to clear secrets from memory

3.4.5 Clearing secret memory regions

VeriFast checks that at the end of each function you have freed all allocated memory (or have passed the ownership to some called function or to the calling function) and that at the end of each block, all memory regions allocated on the stack are present in the symbolic heap before they are deallocated. In a protocol participant implementation where, for example, some secret was generated there is an issue concerning deallocation. The buffer containing the secret is described by a `crypto_chars` chunk, but to release allocated memory (both in the heap and on the stack), it needs to be converted to a `chars` chunk. There is however no way to do this[9] for a secret as this would allow to leak secret bytes into regular program variables or into the program's control flow. The function `zeroize` from Listing 14 allows a protocol implementation to erase its generated and retrieved secrets from memory once the protocol is finished. The fact that this is necessary is actually quite sensible and it can be considered good practice to clear all secrets from memory before control is passed back to the invoker of the protocol.

3.5 Cryptograms as the result of cryptographic primitives

Now we discuss all the definitions to symbolically represent the result of a cryptographic primitive in memory as is required for their contracts. In Section 3.5.1 we define the concept of cryptograms as the symbolic representation of the results of cryptographic primitives. Next in Section 3.5.2, we explain that a cryptogram is characterized with a value of the type of cryptographic bytes `list<crypto_char>`. Then in Section 3.5.3, we continue with a discussion on how cryptograms are represented in memory and we finish in Section 3.5.4 with an explanation of how to compare memory regions that contain cryptograms.

3.5.1 Cryptograms

Before we are ready to specify a contract for some cryptographic primitive, we need a symbolic representation of the result of cryptographic computations. We call these results cryptograms, i.e. instances of the inductive datatype `cryptogram` defined in Listing 15. As mentioned before, we focus here on the generation of random values (i.e. symmetric keys and nonces), hashing and authenticated encryption. So in this scope `cryptogram` needs four constructors, while more constructors are necessary in the full encoding of our extended symbolic model.

The first constructor `cg_sha512_hash` of the inductive datatype `cryptogram` represents hash values and its only parameter `pay` of type `list<crypto_char>` serves to record the (possibly secret) payload that was used to create the hash. The second and third constructor both represent random values and we make a distinction between symmetric keys

[9] A **leak** statement makes VeriFast ignore a `crypto_chars` chunk in the heap during symbolic execution at the end of a code block, but for chunks on the stack this does not work.

```
/*@
inductive cryptogram =
  | cg_sha512_hash         (list<crypto_char> pay)
  | cg_nonce               (int principal, int i)
  | cg_symmetric_key       (int principal, int i)
  | cg_aes_auth_encrypted (int principal, int i,
                           list<crypto_char> pay, list<crypto_char> iv)
;
@*/
```

Listing 15: Cryptograms are the results of cryptographic computations

and random nonces for clarity[10]. Each of these two constructors has two parameters and a `cg_nonce(p,i)` or `cg_symmetric_key(p,i)` cryptogram symbolizes the i^{th} random value generated by principal `p`. Authenticated encrypted messages then, are represented by the fourth constructor `cg_aes_auth_encrypted` and this constructor has four parameters. The first two parameters serve to identify the key that was used and the third parameter records the plaintext that was encrypted. The fourth parameter finally, allows for the same plaintext to be encrypted with the same key to different ciphertexts. This is why an initialization vector is used in symmetric encryption and so the parameter `iv` corresponds to the initialization vector that was chosen to create the ciphertext.

It is instructive to contrast this definition of `cryptogram` with the definition of messages in a regular symbolic model of cryptography. Such a definition is shown in Listing 16 as the inductive datatype `msg`. It is only shown there to compare the two representations and it is not part of our approach. A clear similarity between these two definitions is the representation of random nonces and symmetric keys. An obvious difference between the two is that there is no pairing operator to compose two cryptograms in our extended symbolic model while `msg` has the constructor `msg_pair` to compose messages. Another obvious difference is that `cryptogram`, in contrast to `msg`, has no `msg_data` constructor because plain messages simply have the type `list<crypto_char>` in the encoding of our extended symbolic model. The constructors `msg_hash` and `msg_auth_encrypted` on the one hand and their corresponding constructors of `cryptogram` on the other hand finally, are very similar in both definitions except that they differ in the type for their payloads (and optionally initialization vector). In the definition of `msg` this type is `msg` itself and so the definition of messages is a true recursive definition. The definition of `cryptogram` on the other hand is not recursive since the payload in the constructors `cg_sha512_hash` and `cg_aes_auth_encrypted` have the type `list<crypto_char>`. So in our extended symbolic model, from the definition of `cryptogram` alone, it is not immediately clear how to, for example, encrypt a key or compose two encrypted messages into one message. This will however gradually become clear throughout the rest of this text.

3.5.2 Characterization of a cryptogram

In the contract for a cryptographic primitive, we want to link the contents of a C buffer to some symbolic result produced by that cryptographic primitive (i.e. a cryptogram). As a first step we define the pure function `ccs_for_cg` in Listing 17. This function returns

[10] It also prevents keys from being used as random nonces and vice versa.

```
/*@
inductive msg =
  | msg_data            (list<char> raw_data)
  | msg_pair            (msg fst, msg snd)
  | msg_hash            (msg payload)
  | msg_nonce           (int principal, int i)
  | msg_symmetric_key   (int principal, int i)
  | msg_auth_encrypted (int principal, int i,
                         msg payload, list<char> iv)
;
@*/
```

Listing 16: Messages in classic symbolic models: NOT part of approach

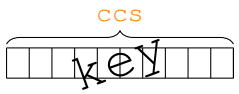
```
//@ fixpoint list<crypto_char> ccs_for_cg(cryptogram cg);
```

Listing 17: Representation of a cryptogram

the characterization for a given cryptogram as a list of elements of type `crypto_char`. It is initially completely unspecified, but its function values are determined during symbolic execution by the postconditions of the cryptographic primitive implementations (as discussed further on). Suppose the primitive for random value generation outputs, in a C buffer, the characterization `ccs` of a symbolic cryptogram `key` as Figure 4 illustrates. Then we know `ccs_for_cg(key)` is equal to `ccs`.

In the encoding of our extended symbolic model, we chose to give this characterization function of cryptograms a surjectivity and an injectivity property. These properties are encoded in Listing 19 and Listing 20 respectively as lemmas about `ccs_for_cg`. The injectivity property's main purpose is to simplify the rest of the approach. The surjectivity property of `ccs_for_cg` does not only simplify the approach, it is also a very useful property when verifying code that parses and interprets raw messages received from the network (see Section 4). Before we discuss these properties of `ccs_for_cg`, we give some straightforward definitions in Listing 18 that allow to differentiate between the different kinds of cryptograms. The inductive datatype `tag` is defined there which has one constructor for each kind of cryptogram. Since none of these constructors has any parameters, we have in fact a unique tag for each constructor of `cryptogram`. The pure function `tag_for_cg` then associates the correct tag with each kind of cryptogram.

In Listing 19 the straightforward surjectivity property of `ccs_for_cg` is expressed. The lemma `ccs_for_cg_sur` simply allows to interpret a list of `crypto_char` as a particular kind of cryptogram. For simplicity, we allow to interpret any list of `crypto_char` (e.g. also the empty list) as the representation of some kind of cryptogram. Since we may assume that



`ccs = ccs_for_cg(key)`

Figure 4: Illustration of the intended meaning of `ccs_for_cg`

```
/*@
inductive tag =
  | tag_hash
  | tag_nonce
  | tag_symmetric_key
  | tag_auth_encrypted
;

fixpoint tag tag_for_cg(cryptogram cg)
{
  switch(cg)
  {
    case cg_sha512_hash(pay1):
      return tag_hash;
    case cg_nonce(p1, c1):
      return tag_nonce;
    case cg_symmetric_key(p1, c1):
      return tag_symmetric_key;
    case cg_aes_auth_encrypted(p1, c1, pay1, ent1):
      return tag_auth_encrypted;
  }
}
@*/
```

Listing 18: Associate a tag which each kind of cryptogram

```
/*@
lemma cryptogram ccs_for_cg_sur(list<crypto_char> ccs, tag t);
  requires true;
  ensures  t == tag_for_cg(result) &*&
           ccs == ccs_for_cg(result);
@*/
```

Listing 19: Surjectivity of the representation of cryptograms

```
/*@
lemma void ccs_for_cg_inj(cryptogram cg1, cryptogram cg2);
  requires tag_for_cg(cg1) == tag_for_cg(cg2) &*&
           ccs_for_cg(cg1) == ccs_for_cg(cg2);
  ensures  col || cg1 == cg2;
@*/
```

Listing 20: Injectivity of the representation of cryptograms

the set of values of the abstract ghost type for secret coin tosses is finite[11], it follows that both sets of possible values for the types `crypto_char` and `list<crypto_char>` are countably infinite. With that assumption then also the set of values of type `cryptogram` is countably infinite. As there are also countably infinite many cryptograms that will never be generated by the cryptographic primitives (due to negative principal identifiers or out of range values in a payload) and since `ccs_for_cg_sur` can map each list that is not generated by a primitive to such a cryptogram, one can conclude that this surjectivity property of `ccs_for_cg` poses no soundness issues.

The injectivity property of `ccs_for_cg` is captured by the lemma `ccs_for_cg_inj` from Listing 20. It expresses that if the characterizations of two cryptograms of the same kind are equal, then the corresponding cryptograms must also be equal or a cryptographic collision occurred. A cryptographic collision, signaled by the boolean function `col`, occurs when during a specific run of a protocol:

- a hash collision is found
- the primitive for random value generation produces the same value twice
- the same ciphertext is computed when the encryption primitive is invoked twice with a different key, plaintext and/or initialization vector as input

Cryptographic collisions should be extremely rare events for well-implemented primitives [12].

3.5.3 Cryptograms in memory

Now we are ready to establish a link between a symbolic cryptogram and an actual C buffer. We do this via a predicate with the overloaded name `cryptogram` defined in Listing 21. The body of this definition reads as follows: "The possibly secret memory region of length `n` starting at address `buffer` is not only correctly allocated, its contents `ccs` is also the characterization of the generated or public cryptogram `cg`." The concept of generated or public cryptograms is important for providing an induction principle for cryptograms which is discussed in Section 3.9.

3.5.4 Comparing secret memory regions

It is clear by now that memory regions that are described by a `crypto_chars` chunk cannot be read. However, some protocols need to compare possibly secret memory regions. Consider for example a protocol where a secret nonce is generated for freshness. One participant generates this value and sends it in an encrypted form to another principal. In some later

[11] This assumption is permissible since an abstract ghost type is not interpreted by VeriFast and we can always find a number of coin tosses which is sufficient for all fixed-length prefixes of a possibly infinite run.

[12] By stating that a collision should be rare, we mean that protocol implementations that force `col` to be true in all symbolic execution branches during verification by using the lemma `ccs_for_cg_inj`, should have a run time complexity that is exponential in the output size of the cryptographic primitive used to force the collision. An example of such an implementation is one that generates so many public random values (i.e. random values with a characterization that can be converted to a value of the type `list<char>`) that the target space gets exhausted; i.e. for random values of n bits, it generates at least $2^n + 1$ random values. As the values of `ccs_for_cg` are only fixed in the postconditions of the cryptographic primitives such exhaustion is the only way to force `col` to be true. To cope with non-terminating protocols, we only consider all finite prefixes of protocol runs.

```
/*@
fixpoint bool cg_is_gen_or_pub(cryptogram cg);

predicate cryptogram(char* array, int n, list<crypto_char> ccs,
                     cryptogram cg) =
  crypto_chars(secret, array, n, ccs) &*&
  ccs == ccs_for_cg(cg) && cg_is_gen_or_pub(cg)
;
@*/
```

Listing 21: Expressing the cryptographic content of a C buffer

stage of the protocol the first principal receives an encrypted message and needs to check that it contains the original value as part of its payload. To do this he needs to compare possibly secret memory regions.

Our approach allows to compare possibly secret memory regions via the standard library function `memcmp`. The contract for this function is shown in Listing 22. It encodes the trivial semantics of the result being equal to zero if and only if the two input buffers have the same content. The pure functions `memcmp_part_ccs`, `memcmp_match` and `memcmp_region`, and the predicate `memcmp_region` are used to ensure that the contract of `memcmp` only accepts comparable memory regions. Two memory regions are comparable if they can be partitioned into an equal number of parts, where corresponding parts have the same size. Moreover, each part must either contain no secrets or must contain exactly the characterization of a single cryptogram so that it is compared in its entirety. This measure ensures that the running time of a badly implemented protocol leaking a secret via `memcmp`, is exponential in the size of that secret. The probability of correctly guessing a secret consisting out of n bits, is 1 in 2^n (assuming a uniform distribution) and all the cryptographic primitives ensure a sufficiently large minimum size for the characterization of each cryptogram. So a principal verified with our model that is implemented to leak, for example, his own secret key is expected to need an exponential number of guesses with `memcmp`.

3.6 Contracts for cryptographic primitives

With all the previous definitions established, we are now ready to give a meaningful contract to a cryptographic primitive. We give contracts for four cryptographic primitives selected from PolarSSL:

- `sha512` (hash generation in Section 3.6.1)
- `havege_random` (random value generation in Section 3.6.2)
- `gcm_crypt_and_tag` (authenticated encryption in Section 3.6.3)
- `gcm_auth_decrypt` (authenticated decryption in Section 3.6.3)

3.6.1 Primitive for hash generation

The first annotated cryptographic primitive we discuss is `sha512` and it is by far the simplest one. It is shown in Listing 23 and is used to generate hash values. The caller of the primitive should provide, besides a valid allocated output buffer, an input buffer expressed by a `crypto_chars` chunk. For simplicity the contract only allows the value 0 for

```
/*@
inductive memcmp_part =
  | memcmp_pub(list<char> cs)
  | memcmp_sec(cryptogram cg);

fixpoint list<crypto_char> memcmp_part_ccs(memcmp_part p)
{
  switch(p){
    case memcmp_pub(cs): return cs_to_ccs(cs);
    case memcmp_sec(cg): return ccs_for_cg(cg);
  }
}

fixpoint bool memcmp_match(list<memcmp_part> l1, list<memcmp_part> l2)
{
  switch(l1){
    case cons(p1, l10): return
      switch(l2){
        case cons(p2, l20): return
          length(memcmp_part_ccs(p1)) == length(memcmp_part_ccs(p2)) &&
          memcmp_match(l10, l20);
        case nil: return false;
      };
    case nil: return l2 == nil;
  }
}

fixpoint bool memcmp_region(list<memcmp_part> l, list<crypto_char> ccs)
{
  switch(l){
    case nil: return ccs == nil;
    case cons(p0, l0): return
      memcmp_part_ccs(p0) == take(length(memcmp_part_ccs(p0)), ccs) &&
      memcmp_region(l0, drop(length(memcmp_part_ccs(p0)), ccs));
  }
}

predicate memcmp_region(list<memcmp_part> l, list<crypto_char> ccs) =
  true == memcmp_region(l, ccs);
@*/

int memcmp(char *array, char *array0, size_t count);
  /*@ requires [?f1]crypto_chars(?kind1, array, ?n1, ?ccs1) &*&
               [_]memcmp_region(?l1, take(count, ccs1)) &*&
               [?f2]crypto_chars(?kind2, array0, ?n2, ?ccs2) &*&
               [_]memcmp_region(?l2, take(count, ccs2)) &*&
              memcmp_match(l1, l2) && count <= n1 &*& count <= n2; @*/
  /*@ ensures  [f1]crypto_chars(kind1, array, n1, ccs1) &*&
               [f2]crypto_chars(kind2, array0, n2, ccs2) &*&
              true == ((result == 0) ==
                       (take(count, ccs1) == take(count, ccs2))); @*/
```

Listing 22: Contract for `memcmp`

```
void sha512(const char *input, size_t ilen, char* output, int is384);
  /*@ requires [?f]crypto_chars(?kind, input, ilen, ?ccs_pay) &*&
               [_]memcmp_region(_, ccs_pay) &*& chars(output, ?olen, _) &*&
               is384 == 0 && olen == 64; @*/
  /*@ ensures  [f]crypto_chars(kind, input, ilen, ccs_pay) &*&
               cryptogram(output, olen, _, cg_sha512_hash(ccs_pay)); @*/
```

Listing 23: A cryptographic primitive to generate hash values

the argument `is384`, so the output buffer must have a size of 64 bytes to store the computed hash value. In the postcondition the predicate `cryptogram` is used to indicate that after an invocation of `sha512`, the output buffer is a memory region linked to the cryptogram `cg_sha512_hash(ccs_pay)`. The fact that the input buffer is expressed by a `crypto_chars` chunk, enables the generation of hash values from possibly confidential data.

The contract of `memcmp` in Section 3.5.4 prevented that a badly implemented protocol could efficiently leak some secret by enforcing that cryptogram characterizations can only be compared in their entirety. To prevent guessing some secret through the payload of a hash, the same mechanism (i.e. the predicate `memcmp_region`) is used to ensure that secret cryptograms can only occur with their entire characterization in the payload. Otherwise a badly implemented protocol can start guessing a secret by comparing a hash of one padded byte of the secret with a hash of a padded known value[13].

An example of how to correctly invoke `sha512` is given in Listing 24. There, a hash value is calculated from the content of the buffer `msg`. First the lemma `chars_to_crypto_chars` is invoked to convert the `chars` chunk that tracks the buffer `msg` to a `crypto_chars` chunk. Subsequently a buffer `hash` is allocated for the primitive to store its computed hash value. After checking that the call to `malloc` succeeded, `sha512` is invoked to compute the hash value from the content of the buffer `msg`. Finally, the heap is cleaned up by clearing and freeing the computed hash value.

3.6.2 Primitive for random value generation

The annotated primitive for random value generation, `havege_random` is shown in Listing 26. To invoke this primitive, an initialized `havege_state` structure is required. One can initialize such a structure with the function `havege_init` from Listing 25. The contract of this function takes a `havege_state` chunk and transforms it into a `havege_state_initialized` chunk. Once all necessary random values are generated, the initialized structure can be discarded via the function `havege_free`, also from Listing 25.

With an initialized `havege_state` structure one can start generating random values. As the precondition of `havege_random` in Listing 26 indicates, a permission to generate random values is required as well. This permission takes the form of a chunk of the predicate `random_permission` as discussed in Section 3.2. The caller of `havege_random` must also provide a `random_request` chunk primarily to pass the ghost arguments `info` and `key_request`. Since we differentiate between keys and nonces, the caller has to indicate if he wants to generate a key or a nonce. He can do exactly this with the parameter `key_request`. The `info` parameter allows to associate some custom information with the resulting cryp-

[13] For encrypted messages this poses no problem as a fresh initialization vector is enforced for each encryption (see Section 3.6).

```
void role1()
  //@ requires principal(?p1, ?random_values);
  //@ ensures  principal(p1, ?new_random_values);
{
  char msg[1024];
  char hash[64];
  /* ... */
  //@ assert chars(msg, 1024, ?cs);
  //@ chars_to_crypto_chars(msg, 1024);
  //@ list<crypto_char> ccs = cs_to_ccs(cs);
  //@ close memcmp_region(nil, nil);
  //@ leak memcmp_region(nil, nil);
  //@ close memcmp_region(cons(memcmp_pub(cs), nil), ccs);
  //@ leak memcmp_region(cons(memcmp_pub(cs), nil), ccs);
  sha512(msg, 1024, hash, 0);
  /* ... */
  //@ crypto_chars_to_chars(msg, 1024);
  //@ open cryptogram(hash, 64, ?hash_ccs, cg_sha512_hash(ccs));
  zeroize(hash, 64);
}
```

Listing 24: Example of generating a hash value

```
struct havege_state{ /* ... */ };
typedef struct havege_state havege_state;

//@ predicate havege_state(havege_state *state) = true /* &*&...*/;
//@ predicate havege_state_initialized(havege_state *state);

void havege_init(havege_state *havege_state);
  //@ requires havege_state(havege_state);
  //@ ensures  havege_state_initialized(havege_state);

void havege_free(havege_state *havege_state);
  //@ requires havege_state_initialized(havege_state);
  //@ ensures  havege_state(havege_state);
```

Listing 25: Initializing and freeing a context for random values

```
/*@ predicate random_request(int principal, int info,
                             bool key_request) = true; @*/
//@ fixpoint int cg_info(cryptogram cg);

int havege_random(void *havege_state, char *output, size_t len);
/*@ requires [?f]havege_state_initialized(havege_state) &*&
             random_request(?principal, ?info, ?key_request) &*&
             random_permission(principal, ?count) &*&
             chars(output, len, _) &*& len >= MIN_KEY_SIZE; @*/
/*@ ensures  [f]havege_state_initialized(havege_state) &*&
             random_permission(principal, count + 1) &*&
             result == 0 ?
               cryptogram(output, len, ?ccs, ?cg) &*&
               info == cg_info(cg) &*&
               key_request ?
                 cg == cg_symmetric_key(principal, count + 1)
               :
                 cg == cg_nonce(principal, count + 1)
             :
               chars(output, len, _);  @*/
```

Listing 26: A cryptographic primitive to generate random values

```
void role1()
  //@ requires principal(?p1, ?random_values);
  //@ ensures principal(p1, ?new_random_values);
{
  //@ open principal(p1, random_values);
  /* ... */
  havege_state state;
  //@ close havege_state(&state);
  havege_init(&state);

  char* key = malloc(16); if (key == 0) abort();
  //@ close random_request(p1, 1234, true);
  if (havege_random(&state, key, 16) != 0) abort();
  /* ... */
  //@ open cryptogram(key, 16, ?key_cs, ?key_cg);
  zeroize(key, 16);
  free(key);
  //@ assert key_cg == cg_symmetric_key(p1, random_values + 1);
  //@ assert cg_info(key_cg) == 1234;

  havege_free(&state);
  //@ open havege_state(&state);
  /* ... */
  //@ close principal(p1, _);
}
```

Listing 27: Example of generating a random key

togram by choosing the function value of cg_info. Function values of cg_info are important when proving integrity properties of protocol implementations (see Section 4). The last requirement imposed by the precondition then, is a correctly allocated output buffer. If the result of a call to havege_random is successful (i.e. the return value is equal to zero), then the postcondition ensures that the content of the output buffer is linked to the proper cryptogram and that this cryptogram has the correct information associated with it. Note that in the postcondition of havege_random the second argument of the random_permission chunk is incremented to ensure that all generated random values are linked with a distinct symbolic cryptogram.

A small example of how to generate a key with all these definitions is shown in Listing 27. First a havege_state structure is initialized and a memory buffer for the key is allocated. Then a random_request chunk is created for generating a key by closing the predicate with the chosen information 1234. Subsequently the actual call to havege_random produces the key in the provided output buffer. After that call, the heap is again cleaned up and two assert statements illustrate some important properties of havege_random. Finally, the havege_state structure is freed.

3.6.3 Primitive for authenticated encryption

The most interesting primitives discussed here are the authenticated encryption and decryption primitives shown in Listing 29 and Listing 30 respectively. As was the case for the primitive for generating random values, also for these primitives some structure must be initialized before they can be invoked. More specifically a gcm_context structure must be initialized with a generated key. This can be accomplished with the function gcm_init from Listing 28. While the gcm module of PolarSSL for authenticated encryption supports multiple ciphers, we chose for simplicity to only write specifications for the AES[14] cipher. This is reflected by the fact that we force the cipher argument of a call to gcm_init to be equal to POLARSSL_CIPHER_ID_AES in the precondition. The precondition also requires, besides a correctly generated key, that the length of that key is 128 bits, 192 bits or 256 bits as is required by the AES cipher. A successful call to gcm_init results in a proper initialized gcm_context structure that later can be cleaned up with the function gcm_free also from Listing 28.

The authenticated encryption primitive gcm_crypt_and_tag is shown in Listing 29. This primitive of PolarSSL can also be directly used for decryption (instead of the one shown in Listing 30), but we chose for conciseness to go for a contract that only allows encryption. Therefore the precondition requires the mode argument to be equal to GCM_ENCRYPT. The precondition also requires the permission to generate random values and this has to do with the 16-byte initialization vector (IV). Since it is important to randomly generate a fresh initialization vector for each encryption, the contract of gcm_crypt_and_tag enforces this using the random_permission chunk. More specifically in the precondition, the assertion iv_cs == chars_for_cg(cg_nonce(p2, c2)) forces the buffer iv to contain a freshly generated nonce and the increment of c2 in the postcondition prevents the same nonce from being reused. The remaining part of the precondition describes the required input and output buffers. Since this primitive performs authenticated encryption, also a buffer to write the

[14] Advanced Encryption Standard (FIPS PUB 197, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf)

```
struct gcm_context{ /*...*/ };
typedef struct gcm_context gcm_context;

//@ predicate gcm_context(gcm_context *context) = true /* &*&...*/;
/*@ predicate gcm_context_initialized(gcm_context *context,
                                      int principal, int count);@*/

int gcm_init(gcm_context *ctx, int cipher,
             const char *key, unsigned int keysize);
  /*@ requires [?f]cryptogram(key, ?size_key, ?cs_key, ?cg_key) &*&
                keysize == size_key * 8 &*&
                cg_key == cg_symmetric_key(?p, ?c) &*&
              gcm_context(ctx) &*&
              cipher == POLARSSL_CIPHER_ID_AES &*&
              (keysize == 128 || keysize == 192
                              || keysize == 256); @*/
  /*@ ensures  [f]cryptogram(key, size_key, cs_key, cg_key) &*&
              result == 0 ?
                gcm_context_initialized(ctx, p, c)
              :
                gcm_context(ctx); @*/

void gcm_free(gcm_context *ctx);
  //@ requires gcm_context_initialized(ctx, _, _);
  //@ ensures  gcm_context(ctx);
```

Listing 28: Initialize or free a context for authenticated encryption

authentication tag or message authentication code (MAC) is required. The implementation of `gcm_crypt_and_tag` also allows some non-encrypted data, identified by the parameters `add` and `add_len`, to be included in the computation of this authentication tag. For simplicity our specification does not support this. The postcondition finally, returns all the updated permissions and links the authentication tag buffer and the output buffer to the correct `cg_aes_auth_encrypted` cryptogram. More precisely, the character representation of the produced symbolic cryptogram `enc_cg` is the concatenation of the authentication tag and the encrypted output. Different choices could be made here, but in the light of our injectivity lemma and our contract for `memcmp` this relieves us from having to enforce a minimum input size for authenticated encryption as is required for normal symmetric encryption (see Appendix A). We also could have chosen to combine the authentication tag buffer and the output buffer into one single continuous buffer and use the predicate `cryptogram` to describe its symbolic content as is done in the contracts of the previously discussed primitives. For flexibility and to support the verification of preexisting code however, we decided to keep these two as distinct buffers. Note that the buffers for the encrypted output, the authentication tag and the IV have become `crypto_chars(secret, _, _, _)` chunks. Indeed, as their updated contents is correlated with the key and input to `gcm_crypt_and_tag`, they could contain secret data[15].

[15] This is actually a crude measure. If both the key and input are not secret, the resulting authentication tag, encrypted message and IV are also not secret. The complete encoding of our extended symbolic model takes this fact into account.

```
int gcm_crypt_and_tag(gcm_context *ctx, int mode, size_t length,
                      const char *iv, size_t iv_len,
                      const char *add, size_t add_len,
                      const char *input, char *output,
                      size_t tag_len, char *tag);
  /*@ requires gcm_context_initialized(ctx, ?p1, ?c1) &*&
               mode == GCM_ENCRYPT &*&
               random_permission(?p2, ?c2) &*&
               crypto_chars(?iv_kind, iv, iv_len, ?iv_ccs) &*&
                 iv_len == 16 &*&  iv_ccs == ccs_for_cg(cg_nonce(p2, c2)) &*&
               add == NULL &*& add_len == 0 &*&
               [?f]crypto_chars(?kind, input, length, ?in_ccs) &*&
               chars(tag, tag_len, _) &*& tag_len == 16 &*&
               chars(output, length, _); @*/
  /*@ ensures  gcm_context_initialized(ctx, p1, c1) &*&
               random_permission(p2, c2 + 1) &*&
               crypto_chars(secret, iv, iv_len, _) &*&
               [f]crypto_chars(kind, input, length, in_ccs) &*&
               crypto_chars(secret, tag, tag_len, ?tag_ccs) &*&
               crypto_chars(secret, output, length, ?out_ccs) &*&
               result != 0 ? true :
                 exists(?enc_cg) &*& cg_is_gen_or_pub(enc_cg) &&
                 append(tag_ccs, out_ccs) == ccs_for_cg(enc_cg) &*&
                 enc_cg == cg_aes_auth_encrypted(p1, c1, in_ccs, iv_ccs); @*/
```

Listing 29: A cryptographic primitive for authenticated encryption

The contract for the primitive for authenticated decryption shown in Listing 30 is very analogous. To decrypt an encrypted message, the IV that was input to encryption and the tag that was output of encryption must be provided. The postcondition simply states that if authenticated decryption was successful and no cryptographic collision occurred, then the correct IV and tag were supplied and the output is the payload of the presented encrypted message. Listing 31 finally, shows a code snippet that correctly encrypts a message. The setting up of the environment for encryption is quite straightforward: a fresh random nonce is generated, it is ensured that the input buffer is described by a `crypto_chars` chunk and a `gcm_context` is initialized. Then the input buffer `msg` is encrypted before the `gcm_context` structure is freed. After the heap is cleaned up, the C function in the example returns.

3.7 Public cryptograms and secret cryptograms

As discussed in Section 3.4, we verify protocols within the random oracle model (ROM) [1] and we assume that all randomness produced by the cryptographic primitives stems from a single source; i.e. a list of coin tosses. We conceptually partitioned this list into public and secret coin tosses and introduced the type `crypto_char` for bytes that may depend on the secret coin tosses and the predicate `crypto_chars` from Listing 8 for such bytes in memory. One of the main objectives of our approach is to prevent the leakage of bytes that depend on the secret coin tosses into regular program variables or into the program's control flow.

As discussed in Section 3.5, the result of a cryptographic primitive is written in a buffer described by a `crypto_chars` chunk. Since the contents of such a chunk may depend on the secret coin tosses, the buffer cannot be read until it is converted to a normal `chars` chunk. If the first argument of a `crypto_chars` is `normal` this conversion is easy as we know there

```
int gcm_auth_decrypt(gcm_context *ctx, size_t length,
                     const char *iv, size_t iv_len,
                     const char *add, size_t add_len,
                     const char *tag, size_t tag_len,
                     const char *input, char *output);
  /*@ requires gcm_context_initialized(ctx, ?p1, ?c1) &*&
               crypto_chars(?iv_kind, iv, iv_len, ?iv_ccs) &*&
                 iv_len == 16 &*&
               add == NULL &*& add_len == 0 &*&
               [?f1]crypto_chars(?kind, tag, tag_len, ?tag_ccs) &*&
                 tag_len == 16 &*&
               [?f2]crypto_chars(kind, input, length, ?in_ccs) &*&
               exists(?in_cg) &*&
               append(tag_ccs, in_ccs) == ccs_for_cg(in_cg) &*&
               in_cg == cg_aes_auth_encrypted(?p2, ?c2,
                                              ?out_ccs2, ?iv_ccs2) &*&
               chars(output, length, _); @*/
  /*@ ensures  gcm_context_initialized(ctx, p1, c1) &*&
               [f1]crypto_chars(kind, tag, tag_len, tag_ccs) &*&
               [f2]crypto_chars(kind, input, length, in_ccs) &*&
               crypto_chars(secret, output, length, ?out_ccs) &*&
               crypto_chars(secret, iv, iv_len, _) &*&
               result != 0 ? true :
                 col || (p1 == p2 && c1 == c2 &&
                         iv_ccs == iv_ccs2 && out_ccs == out_ccs2); @*/
```

Listing 30: A cryptographic primitive for authenticated decryption

is no dependency on the secret coin tosses, but if it is `secret` one first needs to prove that this dependency is absent. We employ an invariant-based approach for these proofs similar to the approach discussed in Section 2.4.

Invariants are the main mechanism in our approach to prove the functional correctness of a cryptographic protocol implementation as in [2, 4, 6]. Instead of enforcing an invariant on public messages on the network, we enforce it on readable memory regions, i.e. memory regions tracked by a `chars` chunk and thus containing no secrets. Since the network API discussed in Section 3.3 only accepts a memory region described by a `chars` chunk, our invariant indirectly holds for all messages on the network. On top of that, our invariant also holds for any memory that can influence the control flow of a protocol implementation. Hence only memory for which the invariant holds can leak to the attacker.

For each verified protocol implementation a custom invariant is required that specifies what cryptographic information is public or, equivalently, non-secret. A good invariant must encode the accumulated knowledge in each message during the execution of a protocol and custom events (see Section 2.3) combined with the information associated with a cryptogram (see Section 3.6.2) are particularly convenient to express this (see the example in Section 4). In our approach cryptograms are the symbolic representation of cryptographic information, so our invariant has to be defined in terms of which cryptograms are public. Listing 32 shows a trivial invariant defined as a the predicate `example_pub`.

For a given invariant, the lemma `public_cg_ccs` in Listing 33 allows to prove that the characterization of a cryptogram that satisfies that invariant indeed does not depend on the secret coin tosses. The lemma `public_ccs_cg`, also from Listing 33, does the opposite. If we

```
void role1(havege_state *state, char *key)
  /*@ requires principal(?p1, ?random_values) &*&
               havege_state_initialized(state) &*&
               [?f1]cryptogram(key, 16, ?key_cs, ?key_cg) &*&
                 key_cg == cg_symmetric_key(?p2, ?id2); @*/
  /*@ ensures  principal(p1, ?new_random_values) &*&
               havege_state_initialized(state) &*&
               [f1]cryptogram(key, 16, key_cs, key_cg); @*/
{
  gcm_context gcm_context;
  char msg[1024];
  char enc[1040];
  char iv[16];
  //@ open principal(p1, random_values);
  /* ... */
  //@ close random_request(p1, 0, false);
  if (havege_random(state, iv, 16) != 0) abort();
  //@ open cryptogram(iv, 16, _, _);
  //@ chars_to_crypto_chars(msg, 1024);

  //@ close gcm_context(&gcm_context);
  if (gcm_init(&gcm_context, POLARSSL_CIPHER_ID_AES, key,
               (unsigned int) 16 * 8) != 0) abort();
  if (gcm_crypt_and_tag(&gcm_context, GCM_ENCRYPT,
                        (unsigned int) 1024, iv, 16, NULL, 0,
                        msg, (void*) enc + 16, 16, enc) != 0)
    abort();
  gcm_free(&gcm_context);
  //@ open gcm_context(&gcm_context);
  /* ... */
  //@ assert exists(?enc_cg);
  //@ crypto_chars_join(enc);
  //@ close cryptogram(enc, 1040, _, enc_cg);
  //@ open cryptogram(enc, 1040, _, enc_cg);
  zeroize(enc, 1040);
  zeroize(iv, 16);
  //@ crypto_chars_to_chars(msg, 1024);
  //@ close principal(p1, _);
}
```

Listing 31: Example of encrypting a message

```
/*@
predicate example_pub(cryptogram cg) =
  switch (cg)
  {
    case cg_sha512_hash(pay0):
      return true /* &*& ...*/;
    case cg_nonce(p0, c0):
      return true /* &*& ...*/;
    case cg_symmetric_key(p0, c0):
      return true /* &*& ...*/;
    case cg_aes_auth_encrypted(p0, c0, pay0, iv0):
      return true /* &*& ...*/;
  }
;
@*/
```

Listing 32: Example of an invariant definition

```
/*@
require_module public_invariant_mod;

predicate public_invar(predicate(cryptogram) pub);

lemma void public_invariant_init(predicate(cryptogram) pub);
  requires module(public_invariant_mod, true);
  ensures  [_]public_invar(pub);

lemma void public_cg_ccs(cryptogram cg);
  requires [_]public_invar(?pub) &*& [_]pub(cg);
  ensures  [_]public_ccs(ccs_for_cg(cg)) &*& true == cg_is_gen_or_pub(cg);

lemma void public_ccs_cg(cryptogram cg);
  requires [_]public_invar(?pub) &*& [_]public_ccs(ccs_for_cg(cg));
  ensures  [_]pub(cg) &*& true == cg_is_gen_or_pub(cg);
@*/
```

Listing 33: Protocol-speficic confidentiality

know that the characterization of some cryptogram does not depend on the secret coin tosses, then it must be so that that cryptogram respects the invariant. When initializing our cryptographic library in the `main` function of the template from Section 3.1, the custom defined invariant must be provided. This is realized by calling the lemma `public_invariant_init` as illustrated by the example in Listing 34. VeriFast's module system is used here again to ensure that this lemma can only be invoked once. Such an invocation of `public_invariant_init` results in a `public_invar` dummy fraction chunk and all threads implementing a protocol participant should receive a dummy fraction of this chunk as also illustrated in Figure 34. This `public_invar` chunk is a necessary permission to call the lemmas `public_cg_ccs` and `public_ccs_cg`.

```
//@ import_module public_invariant_mod;

void role1()
  //@ requires [_]public_invar(example_pub) /* &*& ... */;
  //@ ensures  true;
{ /* ... */ }

/* ... */

int main(void)
  //@ requires module(template, true);
  //@ ensures  true;
{
  //@ open_module();
  //@ public_invariant_init(example_pub);
  // ...
  return 0;
}
```

Listing 34: Initializing the API for proving non-confidentiality

```
//@ fixpoint bool bad(int principal);
```

Listing 35: How to distinguish an honest principal from the attacker

3.8 The attacker model

The strength of the properties proven within our extended symbolic model of cryptography depends on the capabilities of the attacker. If a protocol can withstand a more powerful attacker, it can be considered a more secure protocol. Following the symbolic model our attacker has complete access to the untrusted network. He can grab any message from the network and put any message on there that he can produce using the same cryptographic API as the honest principals. We use the function `bad` from Listing 35 to differentiate honest principals from the attacker.

To allow for the attacker to send anything he can produce with our cryptographic API, he must have the following capabilities:

- Send a part of a message he finds on the network
- Send the concatenation of two messages he finds on the network
- Leak his own generated keys and nonces
- Send a hash created from a message he finds on the network
- Encrypt or decrypt a message from the network with a key he finds on the network and send the result

In the embedding of our extended symbolic model the partitioning and concatenation of messages is trivial and the other capabilities are encoded as lemma function types (for more information on function types in VeriFast see [7]). Listing 36 illustrates such an encoded capability, more precisely the capability of the attacker to leak his own keys. The contract expresses that if a principal is bad, the invariant should hold for the cryptograms representing

```
/*@
typedef lemma void bad_key_is_public(predicate(cryptogram) pub,
                                     predicate() proof_pred)
                                    (cryptogram key);
  requires  proof_pred() &*&
            key == cg_symmetric_key(?p, _) &*& true == bad(p);
  ensures   proof_pred() &*&
            [_]pub(key);
@*/
```

Listing 36: Example of an attacker capabitility

```
/*@
predicate public_invariant_constraints(predicate(cryptogram) pub,
                                       predicate() pred) =
  is_bad_key_is_public(_, pub, pred)
  /* &*& ... */
;
@*/

void attacker();
  /*@ requires [_]public_invar(?pub) &*&
               public_invariant_constraints(pub, ?proof_pred) &*&
               proof_pred() &*&
               principals(?count1); @*/
  /*@ ensures  public_invariant_constraints(pub, proof_pred) &*&
               proof_pred() &*&
               principals(?count2) &*& count2 > count1; @*/
```

Listing 37: The attacker implementation as a C function

his keys. This allows the attacker implementation to convert his own generated keys from `crypto_chars` chunks to `chars` and send them on the network. The custom predicate `proof_pred` can be used to give some extra protocol-specific facts to the proof of the lemma.

Since the invariant for public cryptograms is protocol specific, the proofs that this invariant allows the attacker to perform his attack are also protocol specific. So to ensure that a verified protocol is also capable of withstanding any attack from the attacker (which is the ultimate goal of the entire approach), the invariant must be closed under attacker actions, i.e. the attacker has all the capabilities previously mentioned. This can be proven by writing a lemma implementation for each lemma function type that represents an attacker capability. Only then a chunk of the predicate `public_invariant_constraints` shown in Listing 37 can be created. With such a chunk, the `main` function of the template from Section 3.1, can run the attacker in parallel (i.e. as a separate thread) with your protocol implementation. If the entire application with the attacker as a separate thread still verifies, it is certain that the attacker cannot interfere with the cryptographic protocol.

3.9 Induction principle for cryptograms

In a traditional symbolic model, induction on messages is straightforward. Since the definition of `msg` in Listing 16 of Section 3.6 is a proper inductive datatype, recursive properties of messages can be specified directly through induction. This is required, for example, to specify

```
/*@
fixpoint nat cg_level(cryptogram cg);

fixpoint bool cg_level_below(nat bound, cryptogram cg)
{
  return int_of_nat(cg_level(cg)) < int_of_nat(bound);
}

fixpoint nat cg_level_max();

lemma_auto void cg_level_max_(cryptogram cg);
  requires true;
  ensures  true == cg_level_below(cg_level_max(), cg);
@*/
```

Listing 38: The level of a cryptogram

the invariant for a recursive protocol. In our extended symbolic model the symbolic results of cryptographic primitives are specified by instances of the type `cryptogram`, an inductive datatype which is not recursive. To still allow for recursive reasoning in our model, a custom induction principle for cryptograms was added to the model.

As a first step we associate a level with each cryptogram through the initially completely undefined function `cg_level` from Listing 38. Function values of `cg_level` are natural numbers and they will be determined during symbolic execution by invoking the lemmas discussed further on. An upper bound on the level of a cryptogram can then be expressed via the function `cg_level_below`. The level of a cryptogram corresponds to the length of the longest sequence of recursively nested cryptograms it contains through its payload. It is safe to assume that there exists an upper bound `cg_level_max` for all cryptograms since we only consider finite protocols (or finite prefixes of execution traces of non-terminating protocols) in our approach. In such a finite setting, generated cryptograms cannot be recursively nested in a infinite fashion. The lemma `cg_level_max_` expresses that `cg_level_max` is indeed an upper bound for the level of all cryptograms.

Our induction principle is expressed by the lemma `cg_level_ind` from Listing 39: the level of each generated cryptogram that contains a second cryptogram in its payload, is strictly greater than the level of the second cryptogram. The base case for this induction is implicit as natural numbers are finite.

```
/*@
fixpoint option<list<char> > cg_payload(cryptogram cg)
{
  switch(cg)
  {
    case cg_sha512_hash(pay1):
      return some(pay1);
    case cg_nonce(p1, c1):
      return none;
    case cg_symmetric_key(p1, c1):
      return none;
    case cg_aes_auth_encrypted(p1, c1, pay1, iv1):
      return some(pay1);
  }
}

lemma void cg_level_ind(cryptogram cg, cryptogram cg_pay);
  requires cg_payload(cg) == some(?pay) &*& cg_is_gen_or_pub(cg) &&
           true == sublist(ccs_for_cg(cg_pay), pay);
  ensures  col || true == cg_level_below(cg_level(cg), cg_pay);
@*/
```

Listing 39: Induction principle for cryptograms

4 A Verified Cryptographic Protocol Example

In a moment, we illustrate how to apply our extended symbolic model of cryptography to an implementation of the confidential RPC protocol from Section 2.1. More elaborate verified protocols that illustrate the full capabilities of our approach can be found in the latest VeriFast release and a summary of that verified protocol suite is given in Section 5. Throughout the discussion of the example, we present extracts from the complete implementation of the confidential RPC protocol which is given in Appendix B. In Section 3 we narrowed the scope of our discussion in this text to the generation of random values, hashing and symmetric authenticated encryption. This is also the case for the code extracts displayed here. The version in Appendix B however, shows the same implementation of the confidential RPC protocol that was verified in the complete version of our extended symbolic model.

We discuss all the steps that have to be taken to end up with a verified implementation of the example protocol. First, in Section 4.1, we revisit the protocol transcript of the confidential RPC protocol, restate the security goals using events in the context of our extended symbolic model and encode these goals in the contracts of the functions implementing the protocol roles. In Section 4.2 we present a possible invariant for public messages that is suited for the example protocol and we discuss the rest of the verification process in Section 4.3.

4.1 Encoding the goals of the confidential RPC protocol

The protocol transcript of the confidential RPC protocol is shown in Figure 2. As discussed in Section 2.1, one of the goals of this protocol is the confidentiality of both m_x and $f(m_x)$. This means that after a successful execution of the protocol, only the A and B know the

```
//@ fixpoint bool event_A1(int A, crypto_char m);
//@ fixpoint bool event_B2(int B, crypto_char m);
```

Listing 40: Custom protocol events for example protocol

```
//@ fixpoint crypto_char f(crypto_char m);
//@ fixpoint int A(int p, int c) { return p; }
//@ fixpoint int B(int p, int c) { return cg_info(cg_symmetric_key(p, c)); }

void A_impl(char *k, char *b_m, char *b_fm);
/*@ requires [_]public_invar(example_pub) &*&
             [?f1]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*& principal(A(p, c), _) &*&
             [?f2]crypto_chars(secret, b_m, 1, cons(?m, nil)) &*&
               [_]memcmp_region(cons(_, nil), cons(m, nil)) &*&
             chars(b_fm, 1, _) &*&
             !bad(A(p, c)) && !bad(B(p, c)) && event_A1(A(p, c), m); @*/
/*@ ensures  [f1]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
               principal(A(p, c), _) &*&
             [f2]crypto_chars(secret, b_m, 1, cons(m, nil)) &*&
             crypto_chars(secret, b_fm, 1, cons(?fm, nil)) &*&
             col || (fm == f(m) && event_B2(B(p, c), m)); @*/

void B_impl(char *k);
/*@ requires [_]public_invar(example_pub) &*&
             [?f]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*&
             principal(B(p, c), _); @*/
/*@ ensures  [f]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
             principal(B(p, c), _); @*/
```

Listing 41: Contracts for the roles of example protocol

values of m_x and $f(m_x)$. A second, but less obvious goal is the integrity of the protocol. Participant B only accepts the first message if A indeed has sent it, and participant A only accepts the second message if B sent it.

As discussed in Section 2.3 integrity goals can be expressed through correspondences between protocol events. There, we defined two initial events (i.e. $A_1^{\rightarrow}(m)$ and $B_2^{\rightarrow}(m)$) and to final events (i.e. $B_1^{\leftarrow}(m)$ and $A_2^{\leftarrow}(m)$). To encode these events in VeriFast we only need to explicitly define the initial events as the final events are defined implicitly as termination of the implementation of principal A and B. Listing 40 shows the definition of the events $A_1^{\rightarrow}(m)$ and $B_2^{\rightarrow}(m)$ as the pure functions `event_A1` and `event_B2` respectively. For both these events the identity of the corresponding principal should be provided as the first argument.

Having defined the protocol events in VeriFast, we can now encode the security goals of the confidential RPC protocol. We specify them in the contracts for the functions `A_impl` and `B_impl` that both implement a protocol role. Listing 41 shows the headers of these functions together with their contracts. The uninterpreted pure function `f` represents the response that principal B computes for valid requests and the pure functions `A` and `B` encode the identity numbers of principal A and B given a shared key (see further). It is instructive to first examine the contracts of `A_impl` and `B_impl` while ignoring the security goals.

```
/*@
predicate example_pub(cryptogram cg) =
  switch (cg)
  {
    case cg_sha512_hash(pay):
      return true;
    case cg_nonce(p, c):
      return true;
    case cg_symmetric_key(p, c):
      return bad(A(p, c)) || bad(B(p, c));
    case cg_aes_auth_encrypted(p, c, pay, iv):
      return bad(A(p, c)) || bad(B(p, c)) ?
               [_]public_ccs(pay)
             :
               [_]correct_payload(p, c, pay);
  }
;
@*/
```

Listing 42: Invariant for example protocol

As discussed in Section 3.7, both protocol role implementations require a chunk of the predicate `public_invar` initialized with the protocol invariant (i.e. `example_pub`) which is discussed further on. Each of the roles also needs access to a shared key. The shared key must be generated with the identity of A before any of the protocol roles can be started and the associated information of that key must be the identity of B. If so, using the pure functions `A` and `B`, both principal identities mentioned in the contracts of `A_impl` and `B_impl` correspond to the correct principal. The contract of principal A further requires a memory region `b_m` that contains the request message `m` and this region must comparable with `memcmp` (see Section 3.5.4).

Both the confidentiality and integrity goals of the confidential RPC protocol are encoded in the contract of `A_impl`. Given the request `m` and the fact that the shared key cryptogram `cg_k` equals `cg_symmetric_key(p,c)`, the integrity of the protocol is encoded as follows: if the event `event_A1(A(p,c), m)` occurred upon invocation of principle A, then when `A_impl` terminates the event `event_B2(B(p,c), m)` must have occurred. This implicitly encodes both integrity properties discussed in Section 2.3. Confidentiality is expressed by the fact that buffer `b_m` required by `A_impl` is described by a `crypto_chars(secret, _, _, _)` chunk. So in order to have retrieved the response from principal B as encoded in the postcondition of `A_impl`, A must have communicated the request message to B in encrypted form since A has no means to prove that it is not public and send it in the clear.

4.2 Invariant for public messages

Defining the protocol-specific invariant is the most difficult part of verifying a protocol implementation. The invariant should allow honest principals to execute the protocol and should allow the attacker to perform all the operations that he is capable of according to our attacker model. A possible invariant for the example protocol is shown in Listing 42 as the predicate `example_pub`. It uses the auxiliary predicate `correct_message` defined in Listing 43.

```
/*@
fixpoint bool correct_message(int p, int c, list<crypto_char> pay)
{
  switch(pay)
  {
    case nil:
      return false;
    case cons(m, rest1):
      return switch(rest1)
             {
               case nil:
                 return event_A1(A(p, c), m);
               case cons(fm, rest2):
                 return rest2 == nil && fm == f(m) &&
                        event_B2(B(p, c), m);
             };
  }
}

predicate correct_payload(int p, int c, list<crypto_char> pay) =
  true == correct_message(p, c, pay) &*&
  [_]memcmp_region(cons(_, nil), take(1, pay))
;
@*/
```

Listing 43: Auxiliary constructs for the invariant

The protocol-specific invariant from Listing 42 determines for each cryptogram whether it is public or not. Being able to close a chunk of the predicate `example_pub` indicates that the cryptogram that was provided as an arguments is indeed public. A `cg_sha512_hash` or `cg_nonce` cryptogram is trivially public according to the definition of `example_pub`. This is reasonable for the example protocol as no participant needs to send a hash or nonce on the network, but the attacker must be able to send his computed hashes and nonces as part of his attack. Hence the cases for `cg_sha512_hash` and `cg_nonce` in the definition of `example_pub` simply allow all hashes and nonces on the network. Next, the case for `cg_symmetric_key` cryptograms is defined in terms of the pure function bad discussed in Section 3.8. If any of the owners of the key is bad, the key is public. The case for a `cg_aes_auth_encrypted` cryptogram finally, makes a distinction depending on which key was used for encryption. If the involved key is public, the payload must be public. Since all the keys the attacker has access to are public, this allows the attacker to encrypt or decrypt any message he finds on the network and subsequently put the result back on there. If the used key is not public, a chunk of the predicate `correct_payload` is required.

The predicate `correct_payload` is defined using the pure function `correct_message` in Listing 43. According to `correct_message` a message is correct if it only contains a request `m` and the corresponding event has occurred, or if it also contains a response `f(m)` and the corresponding event occurred. The predicate `correct_payload` does not only enforce that the payload is a correct message as defined by `correct_message`, it also ensures that the first part of the payload is comparable with `memcmp`. This enables principal A to check via `memcmp` that the response it received, was indeed an answer to the pending request.

Protocol	ALOC	SLOC	Ratio	VTime
dummy	130	90	1,44	0.69
hmac	262	132	1,98	0.68
rpc	467	175	2,67	0.81
enc_and_hmac	421	178	2,37	0.81
enc_then_hmac	407	179	2,27	0.84
hmac_then_enc	448	190	2,36	0.90
hmac_then_enc_tagged	436	193	2,26	1.02
hmac_then_enc_nested	567	231	2,45	1.46
auth_enc	287	159	1,81	0.73
sign	346	181	1,91	0.73
nsl	1132	308	3,68	1.07
yahalom	1325	387	3,42	5.81

ALOC = Annotated Lines of Code
SLOC = Source Lines of Code
Ratio = ALOC/SLOC
VTime = Verification time

Figure 5: Results on protocols verified with the described approach

4.3 Verifying the complete protocol implementation

While defining the invariant is the most difficult part of verifying a protocol implementation with our approach, all the steps up until now were preparations for the biggest task: verifying that the protocol role implementations fulfill their contracts using the specified invariant. This is done interactively using VeriFast. With the definitions and lemmas from the cryptographic API, this should be fairly easy if the invariant is properly defined.

5 Results

Using the approach described in this report we were able to verify a significant number of cryptographic protocol implementations. For now we only verified custom-written protocol implementations and we leave tackling preexisting implementations for future work.

Figure 5 shows some statistics about the protocol implementations we verified. The annotation-to-source-code ratios indicate a high annotation effort and this is as expected since we are dealing with intrinsically difficult problems. However, the complexity of the annotations to verify protocol implementations is relatively low (this is a subjective assessment and difficult to quantify). The motivation for this judgment is that most of the complexity is contained in the cryptographic API. Although we have not applied our method to preexisting protocol implementations at the moment, these initial verification efforts give some hope that the approach described in this text is suited to verify functional correctness of preexisting implementations.

Protocol	ALOC	SLOC	Ratio	VTime
high-level API	4898	1656	2.96	33.86
dummy_protocol	127	94	1.35	1.70
secure_storage	214	123	1.74	1.74
secure_storage_asym	215	121	1.78	1.72
rpc	353	189	1.87	1.89
recursive_otway_rees	663	423	1.57	2.02

Figure 6: Results on high-level API and protocols

We also implemented a classical symbolic API on top of the extended symbolic API. This API is very similar to the one in [2] or [6] and we implemented it to get an estimate of the consistency and usability of our extended symbolic API. Some protocol implementations were also written on top of this classical symbolic API and the results of this effort are shown in Figure 6. We do not further discuss the classical API here. Its fully verified implementation is described in [9] and can be found in the latest VeriFast release.

6 Conclusion

In this report we discussed our extended symbolic model of cryptography. The approach presented here is a further development of the method described in [9]. As in [2] and [6] we verify cryptographic protocol implementations, but we target preexisting implementations written in the C programming language. While our approach was developed to verify preexisting implementations, we only verified self-written protocol implementations written against preexisting cryptographic primitive APIs for now. Applying our approach to complete preexisting implementations is left for future work.

Before discussing our approach, we motivated why the traditional symbolic model is insufficient when verifying preexisting cryptographic protocol implementations. Instead, we propose the extended symbolic model of cryptography for verifying preexisting implementations. After a detailed discussion on the definition of this model in VeriFast, we showed a complete verified example. Then we presented some successful verification efforts within our extended symbolic model. These efforts seem to indicate that our approach is well-suited for the verification of preexisting cryptographic protocol implementations. Currently, a formalization of the entire approach is being developed to prove the soundness of the extended symbolic model of cryptography.

7 Acknowledgements.

The research leading to these results has received funding from the EU Horizon 2020 project VESSEDIA under grant agreement n°731453, from the FWO-project (Flemish Research Fund grant) G.0058.13 and from the OT-project (KU Leuven Research Fund grant) OT/13/065.

This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the EU (B-CCENTRE).

References

[1] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *FIRST ACM CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, ACM*. ACM Press, 1993.

[2] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 445–456, New York, NY, USA, 2010. ACM.

[3] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *In 14th IEEE Computer Security Foundations Workshop (CSFW-14*, pages 82–96. IEEE Computer Society Press, 2001.

[4] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 144–158, 2000.

[5] D. Dolev and A. C. Yao. On the security of public key protocols. Technical report, Stanford, CA, USA, 1981.

[6] F. Dupressoir, A. D. Gordon, J. Jurjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 3–17, Washington, DC, USA, 2011. IEEE Computer Society.

[7] B. Jacobs, J. Smans, and F. Piessens. The VeriFast program verifier: a tutorial. 2014.

[8] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1-2):85–128, January 1998.

[9] G. Vanspauwen and B. Jacobs. Verifying protocol implementations by augmenting existing cryptographic libraries with specifications. In *Software Engineering and Formal Methods, 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, pages 53–68. Springer Berlin Heidelberg, September 2015.

[10] F. Vogels, B. Jacobs, and F. Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11(3):1–57, September 2015.

```
/*@
inductive cryptogram =
/*| ... */
  | cg_aes_encrypted (int principal, int i,
                      list<crypto_char> pay, list<crypto_char> iv)
;
@*/
```

Listing 44: Extension for the definition of `cryptogram`

A Regular Symmetric Encryption

In this appendix we discuss a symmetric encryption and decryption primitive of PolarSSL. We explain their semantics in our extended symbolic model as we did for authenticated encryption in Section 3.6. The contracts of the different PolarSSL functions concerned with regular symmetric encryption, are quite similar to those for authenticated encryption. There is however one big difference: the postcondition for regular decryption needs to take into account successful decryption with the wrong key or initialization vector. This situation does not occur with authenticated decryption, as successful authenticated decryption implies that the correct key was provided. As it turns out, this possibility of successfully decrypting with the wrong key or initialization vector renders the semantics of regular decryption significantly more complex.

A.1 Encryption as a cryptogram

Before we can specify a contract for a symmetric encryption and decryption primitive, we need to extend the definition of `cryptogram` from Section 3.6. In order to have a symbolic representation of symmetric encrypted messages, we add the constructor `cg_encrypted`. Listing 44 illustrates this updated definition of `cryptogram` and the argument list of `cg_encrypted` is exactly the same as that for `cg_aes_auth_encrypted`. The surjectivity and injectivity properties of `ccs_for_cg` also hold for `cg_encrypted`.

A.2 Primitives for symmetric encryption and decryption

For the symmetric encryption and decryption primitives, we chose the AES cipher as in Section 3.6. The C functions selected from PolarSSL that are sufficient to encrypt and decrypt with this cipher are:

- `aes_setkey_enc` (initializing an `aes_context` structure with a key)
- `aes_free` (freeing an `aes_context` structure)
- `aes_crypt_cfb128` (encryption and decryption)

As was the case for authenticated encryption, also for regular encryption some context has to be initialized before anything else. Since we chose the AES cipher, this context is an `aes_context` structure. The function `aes_setkey_enc` from Listing 45 initializes such a structure for a given key and after its usage the function `aes_free`, also from Listing 45, can be used to free it.

```
struct aes_context{ /*...*/ };
typedef struct aes_context aes_context;

//@ predicate aes_context(aes_context *context) = true /* &*&...*/;
/*@ predicate aes_context_initialized(aes_context *context,
                                      int principal, int count);@*/

int aes_setkey_enc(aes_context *ctx,
                   const char *key, unsigned int keysize);
  /*@ requires [?f]cryptogram(key, ?size_key, ?ccs_key, ?cg_key) &*&
                keysize == size_key * 8 &*&
                cg_key == cg_symmetric_key(?p, ?c) &*&
              aes_context(ctx) &*&
              (keysize == 128 || keysize == 192
                              || keysize == 256); @*/
  /*@ ensures  [f]cryptogram(key, size_key, ccs_key, cg_key) &*&
              result == 0 ?
                aes_context_initialized(ctx, p, c)
              :
                aes_context(ctx); @*/

void aes_free(aes_context *ctx);
  //@ requires aes_context_initialized(ctx, _, _);
  //@ ensures  aes_context(ctx);
```

Listing 45: Initializing and freeing a context for symmetric encryption

The C function `aes_crypt_cfb128` from Listing 46 implements both encryption and decryption. Which of these two operations is performed is determined by the value provided for the parameter `mode`: `AES_ENCRYPT` or `AES_DECRYPT`. The next thing the precondition requires is a chunk of the predicate `aes_context_initialized`, an initialization vector `iv` with offset zero[16] and an output buffer of size `length`. The remainder of the requirements in the precondition and the entire postcondition depends on the selected `mode`.

If `aes_crypt_cfb128` is invoked for encryption, the rest of the contract looks very similar to that of `gcm_crypt_and_tag` from Listing 29. The precondition ensures that the initialization vector is a fresh random value and it also requires an input buffer to encrypt. The postcondition returns all the permissions from the precondition and ensures that the resulting output buffer is linked with the correct cryptogram. On first sight, the remainder of the contract for decryption is also very similar to the contract for `gcm_decrypt` from Listing 30. Besides that there is no authentication tag required for `aes_crypt_cfb128`, the only clear differences are the chunk of predicate `decryption_pre` in the precondition and the chunk of predicate `decryption_post` in the postcondition. The purpose of these predicate chunks is discussed next.

[16] We chose to annotate the stream cipher primitive `aes_crypt_cfb128` as if it was a primitive for encrypting a single message of any size, without allowing the updated initialization vector to be used for a subsequent encryption. The advantage in doing so, instead of annotating e.g. the cipher block chaining primitive `aes_crypt_cbc`, is that the contracts do not have to deal with the complexity of padding (although it would be perfectly possible to do so).

```
#define AES_ENCRYPT 1
#define AES_DECRYPT 0
#define MIN_DEC_SIZE 10

int aes_crypt_cfb128(aes_context *ctx, int mode, size_t length,
                     size_t *iv_off, char *iv, const char *input, char *output);
  /*@ requires
        mode == AES_ENCRYPT || mode == AES_DECRYPT &*&
        aes_context_initialized(ctx, ?p1, ?c1) &*&
        // AES only supports an iv with a length of 16 bytes
        // only zero offset allowed, not spec'ed for CBF mode
        crypto_chars(?iv_kind, iv, 16, ?iv_ccs) &*&
        u_integer(iv_off, 0) &*&
        chars(output, length, _) &*& mode == AES_ENCRYPT ?
        (
          random_permission(?p2, ?c2) &*&
          iv_ccs == ccs_for_cg(cg_nonce(p2, c2)) &*&
          [?f]crypto_chars(?kind, input, length, ?in_ccs) &*&
            length >= MIN_DEC_SIZE &*&
          ensures
          ( aes_context_initialized(ctx, p1, c1) &*&
            // enforces a fresh IV on each invocation
            random_permission(p2, c2 + 1) &*&
            [f]crypto_chars(kind, input, length, in_ccs) &*&
            // content of updated iv is correlated with input
            crypto_chars(join_kinds(iv_kind, kind), iv, 16, _) &*&
            u_integer(iv_off, _) &*&
            result != 0 ?
              chars(output, length, _)
            :
              cryptogram(output, length, _, ?cg) &*&
              cg == cg_aes_encrypted(p1, c1, in_ccs, iv_ccs) )
        ) : (
          decryption_pre(true, ?garbage_in, ?p2, ?s, ?in_ccs) &*&
          [?f]cryptogram(input, length, in_ccs, ?cg) &*&
            cg == cg_aes_encrypted(?p3, ?c3, ?out_ccs3, ?iv_ccs3) &*&
          ensures
          ( aes_context_initialized(ctx, p1, c1) &*&
            [f]cryptogram(input, length, in_ccs, cg) &*&
            u_integer(iv_off, _) &*&
            crypto_chars(?kind, output, length, ?out_ccs) &*&
            // content of updated iv is correlated with output
            crypto_chars(join_kinds(iv_kind, kind), iv, 16, _) &*&
            decryption_post(true, ?garbage_out,
                            p2, s, p1, c1, out_ccs) &*&
            garbage_out == (garbage_in || p1 != p3 ||
                            c1 != c3 || iv_ccs != iv_ccs3) &*&
            result != 0 || garbage_out ?
              kind == normal
            :
              kind == secret && out_ccs == out_ccs3 )
        ); @*/
  //@ ensures true;
```

Listing 46: A cryptographic primitive for encryption and decryption

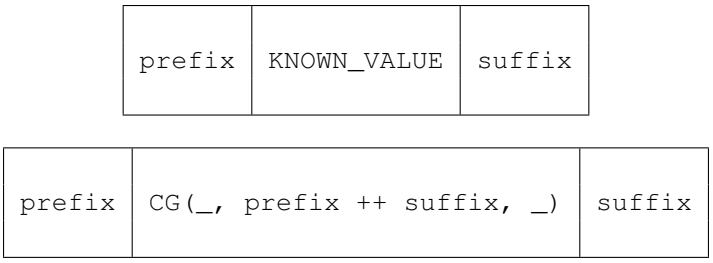


Figure 7: Illustration of the structure of a payload

A.3 Decryption with the wrong key or initialization vector

Before we define the predicates `decryption_pre` and `decryption_post` itself, it is instructive to know what their high-level purpose is. These predicates and all the definitions that follow here, conspire to encode the following observation for unauthenticated decryption:

Successful unauthenticated decryption by itself does not convey any information about the provided key. If however, during some protocol run, one expects the decrypted payload to have a specific structure and it turns out that this expectation was fulfilled, then the encrypted message must have been created with the key or a cryptographic collision occurred.

This observation makes sense as for any secure cipher it should be very hard to construct (without using encryption) a ciphertext that decrypts to a payload with a specific structure. What we mean by a payload having a specific structure is defined next.

Structure of a payload While other and more elaborate interpretations of the concept of structure are perfectly possible, we chose for the interpretation illustrated in Figure 7. Here, two messages are depicted, each with a different kind of structure:

1. The message contains a known value: `KNOWN_VALUE`.
2. Some part of the message is a characterization of a cryptogram that has itself as a payload the concatenation of the rest of the message (i.e. `prefix ++ suffix`).

A protocol participant that wants to perform unauthenticated decryption can express both these kinds of expectations about the resulting payload using the inductive datatype `structure` from Listing 47. To actually prove that a specific payload has some structure, the predicate `has_structure` can be used. This predicate precisely encodes the requirements for a list of `crypto_char ccs` to have the structure `s`. Some minimal sizes are required to ensure that guessing a structure would become infeasible (see further). If one is able to close a chunk of the predicate `has_structure` with the arguments `ccs` and `s`, then one knows that the list `ccs` has the structure `s` according to our model.

Decrypted payload that follows expectation If a protocol participant sees that a decrypted payload fulfills his expectations, then it must have been encrypted with the same key and initialization vector or a cryptographic collision occurred. To encode this fact we add the definitions from Listing 48. A chunk of the predicate `decryption_garbage` should be returned in the postcondition of any unauthenticated decryption function (and thus in the body of the predicate `decryption_post`, see further). As its name suggests, it should

```
/*@
inductive structure =
  | known_value(int offset, list<crypto_char> ccs_known)
  | cryptogram_with_payload(int offset, int length)
;

predicate has_structure(list<crypto_char> ccs, structure s) =
  exists(pair(?prefix, ?suffix)) &*&
  switch(s)
  {
    case known_value(offset, ccs_known):
      return ccs == append(prefix, append(ccs_known, suffix)) &*&
             length(ccs_known) >= MIN_DEC_SIZE &*&
             length(prefix) == offset;
    case cryptogram_with_payload(offset, length):
      return exists(?cg) &*& cg_payload(cg) == some(?ccs_pay) &*&
             ccs == append(prefix, append(ccs_for_cg(cg), suffix)) &*&
             ccs_pay == append(prefix, suffix) &*&
             length(ccs_pay) >= MIN_DEC_SIZE &*&
             length == length(ccs_for_cg(cg)) &*&
             length >= MIN_DEC_SIZE &*&
             length(prefix) == offset;
  };
@*/
```

Listing 47: Describing the structure of some payload

```
/*@
predicate decryption_garbage(bool sym, int principal, structure s,
                             int p_key, int c_key, list<crypto_char> cs_out);

lemma void decryption_garbage(char *b, int n, structure s);
  requires decryption_garbage(?sym, ?p, s, ?p_k, ?c_k, ?ccs) &*&
           col ? true : [_]has_structure(ccs, s);
  ensures  decryption_permission(p) &*& true == col;
@*/
```

Listing 48: A badly decrypted payload has the expected structure

only be returned in the symbolic execution branch of the postcondition where the wrong key or initialization vector was provided. After decryption and once proven that the involved payload has the expected structure, the lemma `decryption_garbage` allows to prove that a cryptographic collision indeed occurred if the wrong key or initialization vector was provided.

For simplicity we left out the concept of key classifiers in Listing 48. To support the complete attacker model from Section 3.8, the lemma `decryption_garbage` should, for some keys, require no proof of structure. A collision is then also not ensured in the postcondition. One example of such keys are the keys of the attacker, since in general he will have no idea what the structure of the decrypted payload will be during his attack. For each protocol implementation, keys are thus classified in two distinct sets by a protocol-dependent key classifier. For one of these sets the lemma `decryption_garbage` has the behavior shown in Listing 48, for the other set it will have the trivial behavior just discussed. The examples in the full encoding of our extended symbolic model illustrate this concept of key classifiers.

```
/*@
predicate decryption_pre(bool sym, bool garbage, int p,
                         structure s, list<crypto_char> ccs_in) =
  !garbage ?
    decryption_permission(p)
  :
    decryption_garbage(sym, p, s, _, _, ?ccs_out) &*&
    exists(pair(?prefix, ?suffix)) &*&
    ccs_out == append(prefix, append(ccs_in, suffix))
;

predicate decryption_post(bool sym, bool garbage, int p,
                          structure s, int p_key, int c_key,
                          list<crypto_char> ccs_out) =
  !garbage ?
    decryption_permission(p)
  :
    decryption_garbage(sym, p, s, p_key, c_key, ccs_out)
;
@*/
```

Listing 49: Predicates `decryption_pre` and `decryption_post`

Decryption permission As discussed in Section 3.2, a principal identity contains amongst other things the permission to perform unauthenticated decryption and this permission plays an important role here. Attentive readers already saw this permission popping up in the postcondition of `decryption_garbage` from Listing 48. This is because after a decryption with the wrong key or initialization vector, the permission should be revoked until one proves that the decrypted payload has the expected structure. So the precondition of any unauthenticated decryption function should require this permission (and thus it should be present in the body of `decryption_pre`, see further). The main reason for temporarily revoking this permission is to prevent a principal from decrypting a message twice, in which case he can use the result of the first decryption to formulate his expectation for the second decryption.

The predicates `decryption_pre` and `decryption_post` Listing 49 shows the definitions of `decryption_pre` and `decryption_post`. We will now explain these predicate definitions step by step. The first parameter of both predicates has the type `bool` and is named `sym`. It indicates if the predicates are used for symmetric or asymmetric decryption. The distinction between these two is not strictly necessary, but it prevents confusion between the contracts for symmetric and asymmetric decryption.

An initial invocation of `aes_crypt_cfb128` is performed with a chunk of the predicate `decryption_pre` where the `garbage` argument is **`false`**. Its body then specifies that a decryption permission is required. As one can see in the definition of `decryption_post`, this permissions is simply returned in the postcondition of `aes_crypt_cfb128`, if the input was no garbage and the provided key and initialization vector were the correct ones. If the input is garbage or if the provided key or initialization vector was not correct, a chunk of the predicate `decryption_garbage` is returned. After checking that the decrypted payload has the expected structure, one can invoke the lemma `decryption_garbage` from Listing 48 to retrieve the decryption permission and to proof that a collision has occurred.

The until now undiscussed, second part of `decryption_pre`, is only relevant for protocol implementations that use some form of nested encryption. Indeed, suppose that in a specific protocol, a participant needs to decrypt a message two times in a row before he can inspect the resulting payload. The definition of the predicate `decryption_pre` allows for this, since in the symbolic execution branch where the first decryption was performed with the wrong key or initialization vector, a chunk of the predicate `decryption_pre` can still be produced for the second decryption. The initial expectation about the structure of the decrypted payload that was fixed in the `decryption_pre` chunk before the first decryption, is simply passed on to the final `decryption_post` chunk after the second decryption. So if the final decrypted payload has the initial expected structure, the lemma `decryption_garbage` from Listing 48 can be invoked to retrieve the decryption permission. Note that it is no problem if the second decryption is only performed on some part of the result of the first decryption. The examples in the latest VeriFast release illustrate this feature.

B Complete verified example

This appendix lists all the source files of the verified protocol implementation discussed in Section 4.

B.1 Header file of the verified protocol implementation

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

#include "polarssl_definitions.h"

#define KEY_SIZE 32
#define PORT     123456

/*@
fixpoint bool event_A1(int A, crypto_char m);
fixpoint bool event_B2(int B, crypto_char m);

fixpoint int A(int p, int c)
  { return p; }
fixpoint int B(int p, int c)
  { return cg_info(cg_symmetric_key(p, c)); }

fixpoint crypto_char f(crypto_char cc);

fixpoint bool correct_message(int p, int c, list<crypto_char> pay)
{
  switch(pay)
  {
    case nil:
      return false;
    case cons(m, rest1):
      return switch(rest1)
             {
               case nil:
                 return event_A1(A(p, c), m);
               case cons(fm, rest2):
                 return rest2 == nil && fm == f(m) &&
                        event_B2(B(p, c), m);
             };
  }
```

```
}

predicate correct_payload(int p, int c, list<crypto_char> pay) =
  true == correct_message(p, c, pay) &*&
  [_]memcmp_region(cons(_, nil), take(1, pay))
;

predicate example_pub(cryptogram cg) =
  switch (cg)
  {
    case cg_nonce(p, c):
      return true;
    case cg_symmetric_key(p, c):
      return bad(A(p, c)) || bad(B(p, c));
    case cg_sha512_hash(pay):
      return true;
    case cg_aes_auth_encrypted(p, c, pay, iv):
      return bad(A(p, c)) || bad(B(p, c)) ?
               [_]public_ccs(pay)
             :
               [_]correct_payload(p, c, pay);
    // following constructors are not described in this technical report
    case cg_sha512_hmac(p, c, pay):
      return [_]public_ccs(pay);
    case cg_aes_encrypted(p, c, pay, iv):
      return [_]public_ccs(pay);
    case cg_rsa_public_key(p, c):
      return true;
    case cg_rsa_private_key(p, c):
      return true;
    case cg_rsa_encrypted(p, c, pay, ent):
      return [_]public_ccs(pay);
    case cg_rsa_signature(p, c, pay, ent):
      return [_]public_ccs(pay);
  };
@*/

void A_impl(char *k, char *b_m, char *b_fm);
/*@ requires [_]public_invar(example_pub) &*&
             [?f1]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*& principal(A(p, c), _) &*&
             [?f2]crypto_chars(secret, b_m, 1, cons(?m, nil)) &*&
               [_]memcmp_region(cons(_, nil), cons(m, nil)) &*&
             chars(b_fm, 1, _) &*&
             !bad(A(p, c)) && !bad(B(p, c)) && event_A1(A(p, c), m); @*/
/*@ ensures  [f1]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
               principal(A(p, c), _) &*&
             [f2]crypto_chars(secret, b_m, 1, cons(m, nil)) &*&
             crypto_chars(secret, b_fm, 1, cons(?fm, nil)) &*&
             col || (fm == f(m) && event_B2(B(p, c), m)); @*/

void B_impl(char *k);
/*@ requires [_]public_invar(example_pub) &*&
             [?f]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*&
             principal(B(p, c), _); @*/
/*@ ensures  [f]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
             principal(B(p, c), _); @*/

/*@
fixpoint bool example_public_key(int p, int c, bool symmetric)
{ return !symmetric || bad(A(p, c)) || bad(B(p, c)); }
predicate example_proof_pred() = true;
PUBLIC_INVARIANT_PROOFS(example)
DECRYPTION_PROOFS(example)
```

```
@*/
#endif
```

B.2 Source file of the verified protocol implementation

```
#include "example.h"

#include <stdlib.h>

#define MSG1_SIZE 33
#define MSG2_SIZE 34


void get_iv(havege_state *state, char *iv)
  /*@ requires [_]public_invar(example_pub) &*&
               havege_state_initialized(state) &*&
               random_permission(?p, ?c) &*& chars(iv, 16, _);@*/
  /*@ ensures  havege_state_initialized(state) &*&
               random_permission(p, c + 1) &*&
               crypto_chars(normal, iv, 16, ?ccs) &*&
               ccs == ccs_for_cg(cg_nonce(p, c + 1));@*/
{
  //@ close random_request(p, 0, false);
  if (havege_random(state, iv, 16) == 0)
  {
    //@ open cryptogram(iv, 16, ?ccs_iv, ?cg_iv);
    //@ close cryptogram(iv, 16, ccs_iv, cg_iv);
    //@ close example_pub(cg_iv);
    //@ leak example_pub(cg_iv);
    // This lemma takes a cryptogram that is public to a chars chunk.
    // It is part of the complete encoding of the extended model of cryptoraphy
    // and can easily be proven with the lemma's discussed in this report
    //@ public_cryptogram(iv, cg_iv);
    //@ chars_to_crypto_chars(iv, 16);
    return;
  }

  abort();
}

void encrypt(havege_state *state, char *k, char *in, int in_size, char *out)
/*@ requires [_]public_invar(example_pub) &*&
             principal(?p0, _) &*&
             havege_state_initialized(state) &*&
             [?f1]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*&
             [?f2]crypto_chars(?kind, in, in_size, ?in_ccs) &*&
             chars(out, in_size + 32, _) &*&
             col || bad(A(p, c)) || bad(B(p, c)) ?
               kind == normal
             :
               [_]correct_payload(p, c, in_ccs); @*/
/*@ ensures  principal(p0, _) &*&
             havege_state_initialized(state) &*&
             [f1]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
             [f2]crypto_chars(kind, in, in_size, in_ccs) &*&
             chars(out, in_size + 32, _); @*/
{
  char iv[16];
  char mac[16];
  gcm_context context;
  //@ open principal(p0, _);

  //@ chars_limits(out);
  get_iv(state, iv);
```

```
  //@ chars_to_crypto_chars(out, 16);
  memcpy(out, iv, 16);
  //@ close gcm_context(&context);
  if (gcm_init(&context, POLARSSL_CIPHER_ID_AES, k,
               (unsigned int) KEY_SIZE * 8) != 0) abort();
  if (gcm_crypt_and_tag(&context, GCM_ENCRYPT, (unsigned int) in_size,
                        iv, 16, NULL, 0, in, out + 32,
                        16, out + 16) != 0)
    abort();
  //@ assert exists(?enc_cg);
  //@ crypto_chars_join(out + 16);
  //@ close cryptogram(out + 16, 16 + in_size, ?enc_ccs, enc_cg);

  /*@ if (!col)
      {
        switch(kind)
        {
          case normal:
            public_ccs(in, in_size);
          case secret:
        }
        close example_pub(enc_cg);
        leak example_pub(enc_cg);
        public_cryptogram(out + 16, enc_cg);
      }
      else
      {
        open cryptogram(out + 16, 16 + in_size, enc_ccs, enc_cg);
        crypto_chars_to_chars(out + 16, 16 + in_size);
      }
  @*/
  //@ crypto_chars_to_chars(out, 16);
  //@ chars_join(out);

  gcm_free(&context);
  //@ open gcm_context(&context);
  zeroize(iv, 16);
  //@ close principal(p0, _);
}

void decrypt(char *k, char *in, char *out, int out_size)
/*@ requires [_]public_invar(example_pub) &*&
             principal(?p0, _) &*&
             [?f1]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*&
             chars(in, out_size + 32, ?in_ccs) &*&
             chars(out, out_size, _); @*/
/*@ ensures  principal(p0, _) &*&
             [f1]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
             chars(in, out_size + 32, in_ccs) &*&
             crypto_chars(?kind, out, out_size, ?out_ccs) &*&
             col || bad(A(p, c)) || bad(B(p, c)) ?
               kind == normal
             :
               kind == secret &*&
               [_]correct_payload(p, c, out_ccs); @*/
{
  char iv[16];
  char mac[16];
  gcm_context context;
  //@ open principal(p0, _);

  //@ chars_limits(in);
  //@ close [1/2]hide_chars(in, out_size + 32, in_ccs);
  //@ chars_split(in, 16);
```

```
  //@ chars_to_crypto_chars(in, 16);
  //@ interpret_auth_encrypted(in + 16, 16 + out_size);
  //@ open [1/2]cryptogram(in + 16, 16 + out_size, ?ccs, ?enc_cg);
  //@ close [1/2]cryptogram(in + 16, 16 + out_size, ccs, enc_cg);
  //@ public_cryptogram(in + 16, enc_cg);
  //@ chars_to_crypto_chars(in + 16, 16 + out_size);
  //@ crypto_chars_split(in + 16, 16);
  //@ assert [1/2]crypto_chars(normal, in, 16, ?iv_ccs);
  //@ assert [1/2]crypto_chars(normal, in + 16, 16, ?mac_ccs);
  //@ assert [1/2]crypto_chars(normal, in + 32, out_size, ?enc_ccs);
  //@ assert ccs == append(mac_ccs, enc_ccs);
  //@ chars_to_crypto_chars(iv, 16);
  memcpy(iv, in, 16);
  //@ chars_to_crypto_chars(mac, 16);
  memcpy(mac, in + 16, 16);

  //@ close gcm_context(&context);
  if (gcm_init(&context, POLARSSL_CIPHER_ID_AES, k,
               (unsigned int) KEY_SIZE * 8) != 0) abort();
  //@ close exists(enc_cg);
  //@ assert gcm_context_initialized(&context, p, c);
  if (gcm_auth_decrypt(&context, (unsigned int) out_size,
                       iv, 16, NULL, 0, mac, 16,
                       in + 32, out) != 0)
    abort();
  //@ assert [1/2]crypto_chars(secret, out, out_size, ?out_ccs);
  //@ open [_]example_pub(enc_cg);
  /*@ if (col || bad(A(p, c)) || bad(B(p, c)))
      {
        if (col)
          crypto_chars_to_chars(out, out_size);
        else
          public_crypto_chars(out, out_size);
        chars_to_crypto_chars(out, out_size);
      }
  @*/
  // Cleanup
  gcm_free(&context);
  //@ open gcm_context(&context);
  zeroize(iv, 16);
  //@ close principal(p0, _);
  //@ crypto_chars_to_chars(mac, 16);
  //@ crypto_chars_to_chars(in, 16);
  //@ crypto_chars_to_chars(in + 16, 16);
  //@ crypto_chars_to_chars(in + 32, out_size);
  //@ chars_join(in);
  //@ chars_join(in);
  //@ open [1/2]hide_chars(in, out_size + 32, in_ccs);
}

void A_impl(char *k, char *b_m, char *b_fm)
/*@ requires [_]public_invar(example_pub) &*&
             [?f1]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*& principal(A(p, c), _) &*&
             [?f2]crypto_chars(secret, b_m, 1, cons(?m, nil)) &*&
               [_]memcmp_region(cons(_, nil), cons(m, nil)) &*&
             chars(b_fm, 1, _) &*&
             !bad(A(p, c)) && !bad(B(p, c)) && event_A1(A(p, c), m); @*/
/*@ ensures  [f1]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
               principal(A(p, c), _) &*&
             [f2]crypto_chars(secret, b_m, 1, cons(m, nil)) &*&
             crypto_chars(secret, b_fm, 1, cons(?fm, nil)) &*&
             col || (fm == f(m) && event_B2(B(p, c), m)); @*/
{
  int socket;
```

```
  havege_state havege_state;

  net_usleep(20000);
  if(net_connect(&socket, NULL, PORT) != 0)
    abort();
  if(net_set_block(socket) != 0)
    abort();

  //@ close havege_state(&havege_state);
  havege_init(&havege_state);
  {
    char msg1[MSG1_SIZE];
    char msg2[MSG2_SIZE];
    char mfm[2];

    /*@ if (col)
      {
        crypto_chars_to_chars(b_m, 1);
        chars_to_crypto_chars(b_m, 1);
      }
      else
      {
        close correct_payload(p, c, cons(m, nil));
        leak correct_payload(p, c, cons(m, nil));
      }
    @*/
    encrypt(&havege_state, k, b_m, 1, msg1);
    //@ open principal(A(p, c), _);
    net_send(&socket, msg1, MSG1_SIZE);
    net_recv(&socket, msg2, MSG2_SIZE);
    //@ close principal(A(p, c), _);
    decrypt(k, msg2, mfm, 2);
    //@ crypto_chars_split(mfm, 1);
    //@ assert crypto_chars(?kind, mfm, 1, cons(?m0, ?rest1));
    //@ switch(rest1){ case cons(x0,xs0): case nil: }
    //@ assert crypto_chars(kind, (void*) mfm + 1, 1, cons(?mf, ?rest2));
    //@ switch(rest2){ case cons(x0,xs0): case nil: }
    //@ chars_to_crypto_chars(b_fm, 1);
    memcpy(b_fm, (void*) mfm + 1, 1);
    /*@ if (col)
        {
          crypto_chars_to_chars(b_m, 1);
          chars_to_secret_crypto_chars(b_m, 1);
          crypto_chars_to_chars(b_fm, 1);
          chars_to_secret_crypto_chars(b_fm, 1);
          MEMCMP_PUB(mfm)
        }
        else
        {
          open correct_payload(p, c, cons(m0, cons(mf, nil)));
        }
    @*/
    //@ open [_]memcmp_region(cons(?p1, nil), cons(m, nil));
    //@ open [_]memcmp_region(cons(?p2, nil), cons(m0, nil));
    //@ append_drop_take(cons(m, nil), 1);
    //@ append_drop_take(cons(m0, nil), 1);
    if (memcmp(b_m, mfm, 1) != 0) abort();
    //@ crypto_chars_join(mfm);
    zeroize(mfm, 2);
  }

  havege_free(&havege_state);
  //@ open havege_state(&havege_state);
  net_close(socket);
}
```

```
void compute_f(char *b_m, char *b_fm)
  /*@ requires [?f]cryptogram(?k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
                k_cg == cg_symmetric_key(?p, ?c) &*&
              crypto_chars(?kind, b_m, 1, cons(?m, nil)) &*&
              chars(b_fm, 1, cons(_, nil)); @*/
  /*@ ensures  [f]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
              crypto_chars(kind, b_m, 1, cons(m, nil)) &*&
              crypto_chars(kind, b_fm, 1, cons(?fm, nil)) &*&
              event_B2(B(p, c), m) && fm == f(m); @*/
{
  //@ open chars(b_fm, 1, cons(_, nil));
  b_fm[0] = 'x';
  //@ open chars(b_fm, 1, cons('x', nil));
  /*@ switch(kind)
      {
        case secret:
          chars_to_secret_crypto_chars(b_fm, 1);
        case normal:
          chars_to_crypto_chars(b_fm, 1);
      }
  @*/
  //@ assume (f(m) == c_to_cc('x'));
  //@ assume (event_B2(B(p, c), m));
}

void B_impl(char *k)
/*@ requires [_]public_invar(example_pub) &*&
             [?f]cryptogram(k, KEY_SIZE, ?k_ccs, ?k_cg) &*&
               k_cg == cg_symmetric_key(?p, ?c) &*&
             principal(B(p, c), _); @*/
/*@ ensures  [f]cryptogram(k, KEY_SIZE, k_ccs, k_cg) &*&
             principal(B(p, c), _); @*/
{
  int socket1;
  int socket2;
  havege_state havege_state;

  if(net_bind(&socket1, NULL, PORT) != 0)
    abort();
  if(net_accept(socket1, &socket2, NULL) != 0)
    abort();
  if(net_set_block(socket2) != 0)
    abort();

  //@ close havege_state(&havege_state);
  havege_init(&havege_state);

  {
    char msg1[MSG1_SIZE];
    char msg2[MSG2_SIZE];
    char mfm[2];

    //@ open principal(B(p, c), _);
    net_recv(&socket2, msg1, MSG1_SIZE);
    //@ close principal(B(p, c), _);
    //@ chars_split(mfm, 1);
    decrypt(k, msg1, mfm, 1);
    //@ assert crypto_chars(?kind, mfm, 1, cons(?m, ?rest1));
    //@ switch(rest1){ case cons(x0,xs0): case nil: }
    //@ assert chars((void*) mfm + 1, 1, cons(_, ?rest2));
    //@ switch(rest2){ case cons(x0,xs0): case nil: }

    compute_f(mfm, (void*) mfm + 1);
    //@ crypto_chars_join(mfm);
```

```
    //@ assert crypto_chars(kind, mfm, 2, cons(m, cons(?fm, nil)));
    /*@ if (!col && !bad(A(p, c)) && !bad(B(p, c)))
        {
          open [_]correct_payload(p, c, cons(m, nil));
          close correct_payload(p, c, cons(m, cons(fm, nil)));
          leak correct_payload(p, c, cons(m, cons(fm, nil)));
        }
    @*/
    encrypt(&havege_state, k, mfm, 2, msg2);
    net_send(&socket2, msg2, MSG2_SIZE);
    zeroize(mfm, 2);
  }

  havege_free(&havege_state);
  //@ open havege_state(&havege_state);
  net_close(socket2);
  net_close(socket1);
}
```

B.3 Main function that executes the verified protocol

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

#include "example.h"
#define NB_OF_RUNS 10

//@ import_module public_invariant_mod;
//@ import_module principals_mod;
//@ import_module decryption_mod;

/*@

predicate_family_instance pthread_run_pre(attacker_t)(void *data, any info) =
  [_]public_invar(example_pub) &*&
  principal(?bad_one, _) &*& true == bad(bad_one) &*&
  public_invariant_constraints(example_pub, example_proof_pred) &*&
  [_]decryption_key_classifier(example_public_key) &*&
  is_public_key_classifier(_, example_pub, example_public_key,
                               example_proof_pred);

@*/

void *attacker_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(attacker_t)(data, ?info);
  //@ ensures  false;
{
  while(true)
    //@ invariant pthread_run_pre(attacker_t)(data, info);
  {
    //@ open pthread_run_pre(attacker_t)(data, info);
    //@ close example_proof_pred();
    attacker();
    //@ open example_proof_pred();
    //@ close pthread_run_pre(attacker_t)(data, info);
  }
  return 0;
}

/*@
predicate_family_instance pthread_run_pre(A_t)(void *data, any info) =
  [_]public_invar(example_pub) &*&
  [1/2]cryptogram(data, KEY_SIZE, ?k_ccs, ?k_cg) &*&
    k_cg == cg_symmetric_key(?p, ?c) &*&
```

```
  principal(A(p, c), _) &*& info == none;

predicate_family_instance pthread_run_post(A_t)(void *data, any info) =
  [1/2]cryptogram(data, KEY_SIZE, ?k_ccs, ?k_cg) &*&
    k_cg == cg_symmetric_key(?p, ?c) &*&
  principal(A(p, c), _) &*& info == none;
@*/

void *A_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(A_t)(data, ?x);
  //@ ensures pthread_run_post(A_t)(data, x) &*& result == 0;
{
  //@ open pthread_run_pre(A_t)(data, x);
  char b_m[1];
  char b_fm[1];

  //@ assert chars(b_m, 1, cons(?m, ?rs));
  //@ switch(rs) { case cons(r0, rs0): case nil: }
  //@ public_cs(cons(m, nil));
  //@ MEMCMP_CCS(cs_to_ccs(cons(m, nil)))
  //@ chars_to_secret_crypto_chars(b_m, 1);
  //@ assert [1/2]cryptogram(data, _, _, cg_symmetric_key(?p, ?c));
  //@ assume (!bad(A(p, c)) && !bad(B(p, c)) && event_A1(A(p, c), c_to_cc(m)));
  A_impl(data, b_m, b_fm);

  zeroize(b_fm, 1);
  //@ public_crypto_chars(b_m, 1);
  //@ close pthread_run_post(A_t)(data, x);
  return 0;
}

/*@
predicate_family_instance pthread_run_pre(B_t)(void *data, any info) =
  [_]public_invar(example_pub) &*&
  [1/2]cryptogram(data, KEY_SIZE, ?k_ccs, ?k_cg) &*&
    k_cg == cg_symmetric_key(?p, ?c) &*&
  principal(B(p, c), _) &*& info == none;

predicate_family_instance pthread_run_post(B_t)(void *data, any info) =
  [1/2]cryptogram(data, KEY_SIZE, ?k_ccs, ?k_cg) &*&
    k_cg == cg_symmetric_key(?p, ?c) &*&
  principal(B(p, c), _) &*& info == none;
@*/

void *B_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(B_t)(data, ?x);
  //@ ensures pthread_run_post(B_t)(data, x) &*& result == 0;
{
  //@ open pthread_run_pre(B_t)(data, x);
  B_impl(data);
  //@ close pthread_run_post(B_t)(data, x);
  return 0;
}

int main(int argc, char **argv) //@ : main_full(main_app)
    //@ requires module(main_app, true);
    //@ ensures true;
{
  pthread_t a_thread;
  havege_state havege_state;

  //@ open_module();
  //@ PUBLIC_INVARIANT_CONSTRAINTS(example)
  //@ public_invariant_init(example_pub);
  //@ decryption_init(example_public_key);
```

```
  //@ DECRYPTION_CONSTRAINTS(example)
  //@ principals_init();
  //@ int attacker = principal_create();

  //@ close havege_state(&havege_state);
  havege_init(&havege_state);
  //@ assume (bad(attacker));
  //@ close pthread_run_pre(attacker_t)(NULL, some(attacker));
  pthread_create(&a_thread, NULL, &attacker_t, NULL);
  //@ leak pthread_thread(_, attacker_t, NULL, some(attacker));

#ifdef EXEC
  int i = 0;
  while (i++ < NB_OF_RUNS)
#else
  while (true)
#endif
    /*@ invariant [_]public_invar(example_pub) &*&
                  havege_state_initialized(&havege_state) &*&
                  principals(?count) &*& count > 0;
    @*/
  {
    //@ int A_id = principal_create();
    //@ int B_id = principal_create();
    char* k = malloc(KEY_SIZE);
    if (k == 0) abort();
    //@ close random_request(A_id, B_id, true);
    //@ open principal(A_id, 0);
    if (havege_random(&havege_state, k, KEY_SIZE) != 0) abort();
    //@ close principal(A_id, 1);
    //@ assert cryptogram(k, KEY_SIZE, ?ccs_k, ?cg_k);
    {
      pthread_t B_thread, A_thread;
      //@ close pthread_run_pre(A_t)(k, none);
      //@ close pthread_run_pre(B_t)(k, none);
      pthread_create(&A_thread, NULL, &A_t, k);
      pthread_create(&B_thread, NULL, &B_t, k);
#ifdef EXEC
      pthread_join(A_thread, NULL);
      pthread_join(B_thread, NULL);
      printf("Iteration %i\n", i);
#endif
      //@ leak pthread_thread(_, A_t, k, none);
      //@ leak pthread_thread(_, B_t, k, none);
     }
    //@ leak malloc_block(k, KEY_SIZE);
  }
}
```